COMPILER-ASSISTED RESILIENCE FRAMEWORK FOR RECOVERY FROM TRANSIENT FAULTS

A Dissertation
Presented to
The Academic Faculty

By

Chao Chen

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology

December 2020

COMPILER-ASSISTED RESILIENCE FRAMEWORK FOR RECOVERY FROM TRANSIENT FAULTS

Thesis committee:

Dr. Santosh Pande
School of Computer Science
Georgia Institute of Technology

Dr. Ling Liu
School of Computer Science
Georgia Institute of Technology

Dr. Greg Eisenhauer
School of Computer Science
Georgia Institute of Technology

Dr. Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Franck Cappello
Mathematics and Computer Science Division
Argonne National Laboratory

Date approved: November 23, 2020

Dedicated to my wife, my daughter and my parents.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisors, Dr. Santosh Pande and Dr. Greg Eisenhauer, for their guidance and supports throughout my PhD studies. It would be impossible for me to reach this point if without your help. You are the one who gave me the freedom to explore various topics and ideas, and gave me suggestions and feedback when I was stuck in troubles. I also yearn for my previous advisor, Dr. Karsten Schwan, who introduced me to the resilience field for high-performance computing systems.

Special thanks go to Dr. Ada Gavrilovska for her guidance during my job search and my days at Gatech. I also want to thank my collaborators, lab mates and all students from the kernel group and the PLSE group. You have helped me in many ways.

Sincere thanks goes to Dr. Ling Liu, Dr. Vivek Sarkar, Dr. Richard Vuduc and Dr. Frank Cappello for serving on my thesis committee. Their insightful feedback has significantly improved this thesis.

I am grateful to Dr. Matthew Wolf. Matt has spend significant amounts of time in my work, and pushed me to think deeper and sharper.

I am also very grateful to the mentors I have had during my internships: Michael K Lang and Latchesar Ionkov from Los Alamos National Lab, Josh Simons and Na Zhang from VMware CTO office.

Finally, I want to thank my parents, my wife and the whole of my family for their support and love. My father Yingshan Chen, my mother Qinggui Cao, my wife Minxiao Sui, and my daughter Anling Chen – you make my dream come true.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Due to system scaling trends toward smaller transistor size, higher circuit density and the use of near-threshold voltage (NTV) techniques, transient hardware faults introduced by external noises, e.g., heat fluxes and particle strikes, have become a growing concern for current and upcoming extreme-scale high-performance-computing (HPC) systems. Applications running on these systems are projected to experience transient errors more frequently than ever before, which will either lead them to generate incorrect outputs without warning users or cause them to crash. Therefore, efficient resilience techniques against transient hardware faults are required for modern HPC applications. Meanwhile, despite increasing threat from transient faults, fault-free operations remain the common case during the execution of applications, thus desirable solutions call for low/no-overhead systems that do not compromise the applications' performance under fault-free conditions.

This dissertation is concerned with the design, implementation, and evaluation of a l͡ight-weight resilience mechanism for large-scale scientific applications to mitigate impacts of transient hardware faults. Under circumstances of transient hardware faults, it aims to help scientists to assure the correct output from large-scale high performance simulations and enable applications to continue their executions upon crashes by repairing corrupted process states automatically and quickly as if there were no faults happened.

In particular, it consists of 3 novel techniques: 1) *LADR*, a light-weight anomaly-based approach to protect scientific applications against SDCs; 2) *CARE*, a low-cost compiler-assisted technique to repair the crashed process on-the-fly when a crash-causing error is detected, such that applications can continue their executions instead of being simply terminated and restarted; and 3) *IterPro*, which targets the problem of recovery from corruptions to the induction variables by exploiting side-effects of modern compiler optimization techniques. To limit the runtime overheads during the normal executions of applications, these approaches exploit properties of scientific applications via compiler techniques.

We comprehensively evaluated the proposed techniques with representative scientific workloads. And the results demonstrate that, attribute to the nature design strategy of these approaches, all of them only incur negligible ($< 3\%$) or even zero runtime overheads during the normal execution of applications, but achieved a high-level fault coverage.

CHAPTER 1
INTRODUCTION

Modern scientific discoveries rely on high performance computing (HPC) systems to run large-scale scientific applications, such as climate modeling, physical simulations, molecular dynamics, and so on, to simulate physical phenomenons. Each run of these applications would take a few days or months even on the world's most powerful HPC system. For these long-running scientific applications, reliability is one of the most important characteristics expected from these high-end computing systems. It is unlikely to efficiently get simulation results if faults are frequently encountered during the run of applications. However, the mean time between failure (MTBF) of current peta-scale systems, which comprises hundreds of thousands of components, is only measured in days or even hours. This is because it is a daunting task to make sure that all components in such large-scale systems are functional at all the times. Although the MTBF of each component is high, the aggregate MTBF of the whole system is low simply because of the large number of system components. The more components the system assembles, the lower MTBF are expected. To mitigate these fail-stop failures, current HPC systems employ checkpoint/restart (C/R) methods [1, 2, 3, 4] to periodically save applications' intermediate states (checkpoints) into a stable storage, and load the latest checkpoint to restart the computation upon a failure. The most popular approach is application-level C/R, where programmers define the states that need to be stored, and instrument applications with specific functions to save essential state and restore from this state in case of failures. C/R has almost become the standard resilience technique in modern production HPC systems.

On the other hand, to meet performance demands of these scientific applications, HPC systems are continuously to exponentially increase the number of system CPU cores. The coming exascale system is projected to be $1000\times$ more powerful and complex than current

peta-scale systems. This is achievable only when supported by the underling manufacturing trends toward higher circuit density, shrinking transistor size and near-threshold supply voltages. However, these factors together would unexpectedly reduce the system MTBF and bring significant challenges to the design of resilience subsystems:

First, the C/R technique may not be suitable for scaling to such large system sizes [5]. C/R is time consuming. Each checkpoint could take as much as $15 \sim 30$ minutes due to limited I/O bandwidth and large data volumes to be saved [3]. Coupled with the requirement that lower MTBF needs higher checkpoint frequency, it implies that majority of computing resources would be "wasted" for doing checkpoints instead of advancing computations, if we continue to use the existing C/R techniques.

Second, transient hardware faults, which are not well considered by C/R techniques, will become more prevalent than expected in future extreme-scale systems [5, 6, 7, 8, 9, 10]. Unlike permanent hardware faults, which typically exhibit visible symptoms, transient hardware faults occur in a non-deterministic way. Therefore, it is challenging to predict when and where a transient hardware fault will occur. Although the error correcting codes (ECC), such as SECDED and chipkill, could help to detect or correct these type of errors in main memories incurring necessary cost in terms of energy and performance, there still lack of efficient techniques against transient hardware faults manifested inside the computing logic.

As there are significant number of studies targeting for optimizing C/R techniques [3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] with new memory devices or software stacks, this dissertation mainly focuses on resilience techniques against transient hardware faults, which is not well supported by existing resilience techniques.

1.1 Transient Hardware Faults and Their Threats

Transient hardware faults are essentially caused by external noises, e.g., cosmic radiations, heat fluxes and particle strikes. Upon occasional interactions with silicon dies, these high-

energy neutrons are very likely to create a secondary cascade of charged particles, which would create current pulses and change (bit-flip) values stored in the memory or produced by computational logic. Smaller transistors are more easily to be flipped because they carry smaller charges, thus less energy is required to flip its value [8]. While such upset events sound to be relatively infrequent in each individual component, they are more frequent than expect in large-scale HPC systems, simply because of the sheer number of assembled components [15, 8]. Their failure rate is expected to increase exponentially according to the system scale size.

Oliveira et al. [10] projected that a hypothetical exa-scale machine built with $190,000$ cutting-edge Xeon Phi processors would experience daily transient errors even though their memory areas are protected with ECC (Error-Correcting Code) techniques. Unlike permanent hardware failures, transient faults only have a temporary impact on hardware, so the hardware can continue the normal operations when flushed with new data. Because of their short duration, it is quite difficult to mask these non-memory faults at the hardware level in a cost-efficient manner [16, 17]. Therefore software resilience solutions are required for modern scientific applications running on these systems.

Depending on where they occur, transient faults mainly introduce two types of problems to modern scientific applications, including: 1). Silent Data Corruption (SDC) which means applications can finish their executions "successfully" but with incorrect outputs, and 2). Crashes (soft failures) which means applications are terminated unexpectedly due to the transient faults. While the transient faults manifested inside Floating-Point Units (FPU) are more likely to lead scientific applications to generate incorrect outputs (SDCs) without any user-aware warnings, the transient faults that corrupt operations in Arithmetic and Logic Units (ALU) will more likely to cause execution crashes because of the address and branch computations performed there. Li et al. [18, 19, 20, 21] quantitatively classified these impacts on scientific applications by leveraging empirical fault injection experiments. They mainly focused on transient faults manifested from the CPU logic assuming

that memory is protected with ECC. Based on studying a set of representative scientific applications, they pointed out that, SDC and soft failures are two major negative outcomes of transient faults: more than 30% of transient faults were manifested as SDCs with an almost equivalent number of them leading to soft failures. Majority of the rest faults simply vanished without causing any impacts to the runs of applications.

1.2 The State of the Art

Current computing systems mainly rely on hardware-level technologies to provide resilient computing services against transient hardware faults. "safety-margin" is firstly leveraged to increase resilience ability of transistors against external noises, e.g., high-energy particles. "Safety-margin" is created by operating a circuit at safe supply voltages and clock rates (e.g., higher operating voltage and lower clock rates), therefore increases the energy budget that is required to flip bits in transistors. As a result, it would enable the hardware to operate correctly even in the face of some variations. To counter against worst case, this design strategy would impose expensive power and performance overheads, which is not feasible for large scale systems. Furthermore, the modern manufacturing trends toward smaller feature size and near-threshold voltage supplies are also breaking this design strategy due to physical space limitations.

Beside the "safety-margin", special radiation resilient materials are also used to insulate transistors from external noises. However, due to high costs, manufacturing productivity constraints and degraded performance, we only saw limited adoption of radiation-hardened devices in mission-critical systems such as satellites, spacecrafts and so on.

In commercial computing systems, hardware-based redundancy techniques are commonly applied. They leverage either the replication of hardware components (e.g. triple modular redundancy) or redundant information (e.g. error correcting code) to detect and correct errors transparently. IBM has historically added 20%$\sim$30% additional logic unit for fault tolerance, and fully replicated processors' execution units when designing S/390

G5 [22, 23]. Such hardware redundancy approach is very expensive in terms of both hardware overheads and power consumption. On the other hand, ECC [24], an information redundancy approach, is widely adopted in current production systems to protect the main memory subsystem. However, it has not been applied to the CPU unit, in which transient hardware faults are projected to become significant. Intel estimated that additional correctness checks on chip will increase the power consumption by $15 \sim 20\%$, which is not viable for extreme scale HPC systems due to limited power budgets.

These limitations motivated the demands of software-based techniques for future extreme scale HPC systems and scientific applications. As compared to the hardware techniques, software approaches are more flexible and effective. They can be easily configured per application properties and requirements. In addition, they do not require hardware modifications, which is a good fit to HPC systems built with commercial hardware components.

1.3 Challenges for Building Software-based Resilience Solutions

Although there are many software-based approaches proposed in past years, the resilience issue has not been addressed yet, and there are still significant challenges to be addressed for efficient software-based resilience solutions.

1. **High-level fault coverage should not compromise applications' performance**. Despite increasing threat from transient faults, fault-free operations remain the common case during the execution of parallel applications. Therefore, it is unwise to significantly degrade applications' under normal circumstances when there are no faults. However, majority of existing techniques seek to detect and correct SDCs by duplicating computations through either executing the same copy of applications on different sets of hardware [4, 25, 26] or duplicating the instructions of applications via compiler techniques [27, 28]. They would require $\sim 2\times$ resources overheads, which implies that such techniques have seen limited adoption in scientific computing. Meanwhile, algorithm-based fault tolerance techniques are also emerging as

alternatives. They focus on designing resilient data structures and algorithms. However, these algorithm-specific methods are highly specialized and as such focus on specific computational kernels, which are normally a small part of scientific applications. Faults that happen outside the computational kernel but still affect the core computation may be not detectable in such cases, thus would lead to low fault coverage.

2. **There lacks of efficient methods for recovering from soft failures**. While there are a bunch of studies for detection and correction of SDCs, less research effort has been spent on handling soft failures, perhaps because the community takes it for granted that the standard Checkpoint/Restart (C/R) methods can provide adequate recovery. Unfortunately, while the C/R technique can help scientific applications to recover from soft failures, it is also very costly in terms of lost opportunities (batch job slots), lost computation (everything since the last checkpoint) and I/O overheads (repeatedly writing checkpoint files). These costs are particularly significant for massively parallel jobs [3, 5, 26] and also suffer from very poor scaling effects. LetGo [20] is an exception which is specially designed to continue the execution of an application upon soft failures. It relies on a set of heuristics to patch the corrupted process states. For example, it assumes that the result of an failed instruction is 0, if the instruction is to access an invalid memory address at low address. Obviously, such kind of technique is not to repair the corrupted states, but to convert a soft failure into a SDC, which is another challenge problem for scientific applications.

1.4 Thesis Statement

Motivated by the above observations and challenges, this dissertation addresses the following thesis statement:

A compiler-assisted resilience mechanism that explores properties of scientific applications is a viable way to assure the correct output from large-scale high performance simula-

tions and continue the execution of applications by repairing crashed processes on-the-fly with negligible overheads.

In order to demonstrate this statement, the thesis identifies the technical challenges in designing and implementing resilience mechanisms against transient errors, develops compiler-assisted solutions to address those challenges, applies the solutions to multiple representative scientific workloads, and experimentally evaluates their benefits based on empirical fault injection experiments.

1.5 Contribution

In this dissertation, we make the following major contributions.

1. We proposed LADR, a lightweight SDC detection technique for scientific applications. LADR protects scientific applications from SDCs by watching for data anomalies in their state variables (those of scientific interest). It employs compile-time data-flow analysis to minimize the number of monitored variables, thereby reducing runtime and memory overheads while maintaining a high level of fault coverage with low false positive rates. We designed and implemented the LADR based on the LLVM framework. It supports a majority of scientific applications written in C/C++ and Fortran. Despite some limitations that need to be refined, our prototype of LADR still presents one step towards building application-level SDC mitigation frameworks.

2. We studied the manifestation of soft failures in modern scientific applications through empirical instruction-level fault injection experiments. We classified the soft failures based on hardware trap symptoms, and examined their manifestation latency measured in terms of number of dynamic instructions. We found that: 1). majority of soft failures (as much as 98.95%, and 91.45% on average) evidence themselves by causing a *SIGSEGV* because of invalid memory access; and 2) most soft failures would

manifest within a few dynamic instructions. Hence, the original raw data used for array location computations remains uncontaminated, and can be used to recompute the corrupted state. Therefore, we designed and implemented CARE, a light-weight compiler-assisted technique to repair soft failures on-the-fly when a crash-causing error is detected, such that applications can continue their executions instead of being simply terminated and restarted. During the compilation of applications, CARE constructs a recovery kernel for each crash-prone instruction, and upon an occurrence of an error, CARE attempts to repair corrupted state of the process by executing the constructed recovery kernel to recompute the memory reference on-the-fly.

3. We presented IterPro, which augments CARE by exploiting side effects introduced by code optimization techniques such as strength reduction and loop unrolling, which are widely adopted by modern compilers. While these code optimization techniques were mainly designed to improve the execution speed, they introduce equivalent computation patterns and values (semi-redundancies) into the code, providing opportunities which can be exploited for resilience purposes. IterPro leverages these introduced semi-redundancies to repair the crashes due to corruptions in induction variables, which are otherwise unrecoverable.

As a body of work, this dissertation demonstrates a compiler-assisted solution to building lightweight resilience mechanisms against transient errors. The solution incurs negligible runtime overheads while maintaining a high-level of fault coverage.

1.6 Organization of the Dissertation

The rest of the dissertation is organized as follows:

- Chapter 2 presents an overall picture of proposed software-based resilience mechanism against transient hardware faults.

- Chapter 3 presents a lightweight SDC detection technique by exploring the data-similarity feature and data-flow information of scientific applications.

- Chapter 4 studied manifestation of soft failures based empirical fault injection experiments. The insights from this study motivated the design of a light-weight failure recovery framework that can repair the crashed process on-the-fly with almost zero runtime overheads, such that impacted applications can continue their executions as normal as if there were no faults. It also exploits equivalent computation patterns introduced by modern compiler optimization techniques for recovery from failures caused by corruption in induction variables.

- Chapter 5 finally summarizes the dissertation, draws conclusions, and discusses open problems and future research directions.

CHAPTER 2

SYSTEM OVERVIEW

This dissertation presents a compiler-assisted resilience framework protecting large-scale scientific applications from transient hardware faults. The framework will help to validate the results of the simulations to be free of SDCs and repair the process crashes caused by transient hardware faults such that applications can continue their executions normally instead of being simply terminated and restarted. Therefore, it can improve the efficiency of modern scientific discovery processes. The proposed framework exploits applications properties to minimize the overheads involved in detection and recovery. To achieve this goal, the framework consists of two components, a compiler front-end for performing program analysis and extracting applications' properties, as well as a runtime system consuming the information generated by the compiler front-end and providing resilience services. Figure 2.1 depicts an overall picture of the framework.

The compiler front-end consists of a set of compiler passes, which perform necessary code transformations to explicitly export code features for resilience purpose, and program analysis to comprehensively identify, extract and convey these features to the underlying runtime system. And the runtime system of the proposed framework is presented as a set of shared libraries, which will be linked to applications and provide resilience services for SDC detection and soft failure recovery as guided by the information generated by the compiler component.

Based on the functionality of each component, the framework can be further divided into two separate and independent sub-modules. In particular, `libProtector.so` and `Analyzer` form a sub-module named LADR, which provides functionality of SDC detection; `libSafeguard.so` and `Armor` formed another sub-model called CARE that is to provide the recovery service against soft failures. These two sub-modules work indepen-

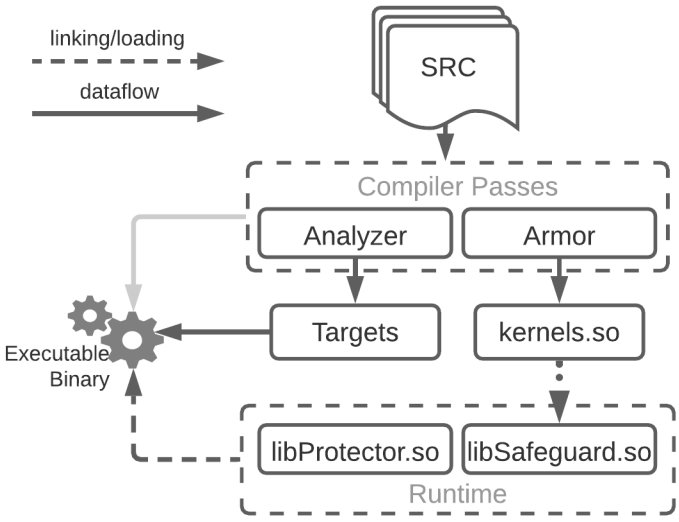


Figure 2.1: Overview of the Compiler-Assisted Resilience Framework against Transient Hardware Faults

dently.

For SDC detection, the application is compiled with the `Aanlyzer` pass, which performs the program analysis to build the data-flow graph among state variables (arrays) of scientific applications. Based on the data-flow graph, a small subset of state variables (indicated by the `Targets` in Figure 2.1) can be identified such that SDCs in other variables are finally propagated to this subset of variables. The `libProtector.so` library implements an interface, which is called inside the code to monitor the value changes in the selected state variables. It detects SDCs by monitoring value change patterns of each data points in monitored state variables. This detection strategy is motivated by the observation that scientific applications exhibit data-similarities across time steps. Different data similarity algorithms are implemented in our prototype. To further reduce the runtime overhead, `libProtector.so` divides the data points for each monitored state variable into several groups, with each group containing a few data points. For each group, a feature value, e.g., a representative mean, standard deviation, or mass is extracted, which forms a new and much smaller feature array. The framework thus works on the feature instead of original data points for SDC detection, which significantly reduces the number monitored

data points, therefore reduces the runtime overhead. The `libProtector.so` employs an algorithm to dynamically divide data points into groups based on features of the data to improve the detection efficiency. Currently, when a SDC is detected by LADR, the process is terminated, and the checkpoint/recovery mechanism is activated rolling back the application to a saved error-free state.

On the other hand, for soft failure recovery, `Armor` and `libSafeguard.so` are involved. `Armor` itself consists of 3 independent compiler passes. These passes work in a predefined order to perform necessary code transformations and program analysis to build a set of recovery kernels for failure-prone instructions. It then compiles these recovery kernels into a standalone shared library (represented as `kernels.so` in Figure 2.1). `Armor` constructs recovery kernels by strategically cloning related instructions from original source codes, and each recovery kernel is represented as a function in a similar way to the C language. Upon a failure detected by operating system, a method in `libSfeguard.so` is automatically activated to diagnose the failure, disassemble the failed instruction and find the (potentially) corrupted architecture state, load the recovery kernels library (`kernels.so`), search for the appropriate recovery kernel, and execute it to get a new memory access address. If the new address is the same as the current accessed address, the recovery has failed and the process is terminated. Otherwise, it will repair the related architecture state with a value derived from the new address, and the process will re-execute the failed instruction, and continue the execution as normal.

CHAPTER 3
LIGHT-WEIGHT SDC DETECTION

In this chapter, we present LADR, a lightweight SDC detection technique for scientific applications. It shares a similar detection model to techniques introduced in [29, 30] in that they all detect SDCs by watching for data anomalies in state variables of scientific applications. Built on top of this philosophy, LADR employs compile-time data-flow analysis to minimize the number of monitored variables, thereby reducing runtime and memory overheads while maintaining a high level of fault coverage with low false positive rates. On the evaluated benchmarks, LADR achieved $> 80\%$ fault coverage with only $\sim 3\%$ runtime overheads and $\sim 1\%$ memory overheads. We believe that such an approach with low memory and runtime overheads coupled with attractive detection precision is a viable approach for assuring the correct output from large-scale high performance simulations

3.1 Introduction

As discussed in chapter 1, SDC is one of major negative impact of transient hardware faults for scientific applications, and it has attracted intense attentions. Complete and comprehensive SDC detection requires the duplication of computing through either hardware or software redundancy [25, 4, 31], but the overheads required ($\sim 2\times$ resources), and the fact that fault-free operations remain the common case despite the increasing transient fault threat [32], mean that such techniques have seen limited adoption in scientific computing. Fortunately, many scientific applications can tolerate some errors in their outputs [33, 10], as long as the errors don't introduce new data features. This allows the HPC community to trade-off fault coverage for the performance, and has motivated the development of anomaly-based detection methods [34, 35, 30]. These methods seek to exploit characteristics implicit in many scientific applications, such as those which iteratively simulate

changes in physical properties, in order to determine when a calculated value has fallen outside its 'expected' range based upon either prior or neighboring values. It is worth to note that while applying such techniques to **all** data in an application would be impractical, it is viable to limit its use to only "variables of scientific interest" [30] (e.g., variables that are actually output for further analysis, or that are used in checkpoint/restart. We'll call them the 'crucial variables'.) with relatively low overheads. However, many scientific applications i.e., Parallel Ocean Program (POP), have dozens of crucial variables, and even when these techniques are limited to that set of data, overhead imposed are still significant, perhaps too high for the techniques to gain acceptance by the scientific community.

LADR is therefore designed to reduce these overheads by exploring correlations among crucial variables and data points. LADR shares a similar detection model to that in [30, 29]. It is built upon those prior works by introducing compiler techniques to detect and utilize the data-flow of the application in order to limit monitoring overhead. LADR reduces the overheads primarily through minimizing the number of monitored variables. For a given set of crucial variables C, LADR monitors a smaller set of variables D, such that if variables in C are contaminated by SDCs, variables in D will also be contaminated; or put differently, SDCs will be definitely propagated from C to D. We name variables in D as **sink variables** of C. LADR finds D mainly leveraging compile-time data-flow analysis. The similar idea was also presented in [29], but without reference to compile-time analysis. In addition, for each variable in D, LADR also applies a data grouping technique to further reduce runtime and memory overheads, which is not explored in prior studies [30, 29]. LADR is designed to detect SDCs in crucial variables. SDCs in control variables are not covered unless they are propagated as corruptions to crucial variables.

In this chapter, we mainly made the following contributions:

1. we proposed a methodology to minimize runtime and memory overheads for anomaly-based SDC detection techniques.

2. we designed and implemented the LADR based on the LLVM framework, and sup-

ports a majority of scientific applications written in C/C++ and Fortran. Despite some limitations that need to be refined, our prototype of LADR still presents one step towards building application-level SDC mitigation frameworks.

3. We evaluated LADR with 4 representative scientific workloads including GTC-P, POP, LAMMPs and miniMD. We find that LADR is able to protect them from influential SDCs with as low as $\sim$ 1% memory overheads, and $\sim$ 3% runtime overheads. As compared to the state-of-the-art anomaly-based detection technique, LADR achieved comparable faulty coverage, but reduced runtime overheads by more than 20% and memory overheads by up to 55%.

The evaluation results suggest that LADR is a promising solution for scientific applications that can tolerate small numerical fluctuations in their outputs. Certainly, LADR has its constraints for some applications or in some operational situations in which the accuracy of results is the users' primary concern. We believe, however, LADR is attractive in many situations since many scientific simulations themselves are approximate computing to physical phenomenons and can tolerate some small errors [33, 36, 37] in their output.

The rest of the chapter is organized as follows: section 3.2 introduces relevant backgrounds and observations that motivated this work; section 3.3 depicts the overall design of LADR; section 3.4 presents the design details of LADR for identifying sink variables; section 3.5 describes the runtime design of LADR; section 3.6 and section 3.7 respectively present the prototype and evaluation results. Next, related works are discussed in section 3.8, followed by conclusion in section 3.9.

3.2 Background and Motivation

LADR is built on top of the state-of-the-art data-anomaly SDC detection methods. It aims to optimize these methods by exploiting applications' properties. Therefore, in this section, I will first present the philosophy of the data-anomaly SDC detection; and then present the

observation that motivated the design of LADR.

3.2.1 Anomaly-based Techniques

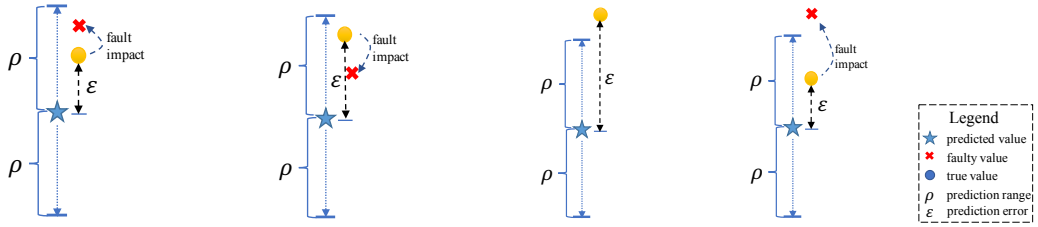
Anomaly-based detection methods mainly exploit the characteristics of the application data to detect SDCs. Yim et al. [38] computed the histogram of application data to detect outliers in conjunction with temporal and spatial similarity. Di et al. [30] characterized features of applications' outputs, and exploited the smoothness of their outputs across time dimension for detecting SDCs.

These techniques generally have three phases: 1) predicting the next expected value in the time series for each data point; and 2) determining a bound, e.g. normal value interval, surrounding the predicted value. 3) detecting possible SDCs by observing whether the observed value falls outside the bound. There are 4 possible situations as depicted in Figure 3.1. Obviously, as compared with complete computational redundancy, these methods trade off fault coverage for performance. They could miss SDCs if the selected bound ρ (shown in Figure 3.1(a)) or the prediction error ε (shown in Figure 3.1(b)) is larger than the impact of SDCs. On the other hand, it would incur false positives if ρ is smaller than ε. A case depicting a successful detection is shown in Figure 3.1(d).

As mentioned before, our work shares a core similarity to that of Di et al. [30] with respect to the basic approach of detecting SDCs through analysis of predicted values, and Berrocal et al. [29] introduces the concept of exploiting correlation between variables. LADR extends these prior work by proposing compiler techniques to exploit dataflow-based variable correlation, and in exploring point grouping strategy for additional overhead reduction.

3.2.2 Propagation of SDCs

Although by no means universal, time-step based iterative solvers are an important component in many scientific simulations, with each time-step implementing a time point in



(a) false negative due to large ρ

(b) false negative due to high ε

(c) false positive

(d) valid and detectable

Figure 3.1: Detection model of anomaly-based techniques (figure derived from [30])

the real world. These applications simulate real-world phenomena by solving a system of differential equations on points (e.g., grids, particles). Each point could be associated with several *states* (i.e., speed, temperature, energy etc.), which are crucial variables of applications. Applications proceed along the temporal dimension to update *states* for each point. The result of each time-step will be taken as the input to the next time-step. Climate models are typical examples. They treat the atmosphere as a cubic box divided into grids, and each grid represents a geometrical area on the earth. Climate parameters, e.g., temperature, are computed by solving atmosphere dynamic equations for each grid. To update a specific state for a grid, the values of other states of other grids would be used. Due to this nature, these applications present two general characteristics that inspired the design of LADR.

First, SDCs propagate among crucial variables during the iterative updates. For example, Table 3.1 shows that, for GTC-P and POP, the contamination of multiple crucial variables could result from a single fault injected into a data element of crucial variable listed in the first column. Therefore, it is possible to detect SDCs by only monitoring a subset of crucial variables. For GTC-P, specifically, it is possible to protect the variables $z0$, $z1$, $z2$, $z3$, $z4$, and $z5$ by only monitoring the variable $moments$.[1] In observing this, LADR shares a similarity with [29], which also notes that error correlation between variables can be exploited to reduce SDC monitoring overhead, but does not specially propose

[1]Each of these variables store attributes of particles, e.g., weight and velocity. The names are as they appear in GTC-P source code.

Table 3.1: Correlation among crucial variables. It is based on controlled fault injection experiments. Faults are injected to variables in the first column and then the outputs of other variables are checked against the output of fault-free run. Each run of the application performs one injection to one variable.

(a) GTC-P Particle Data Variables

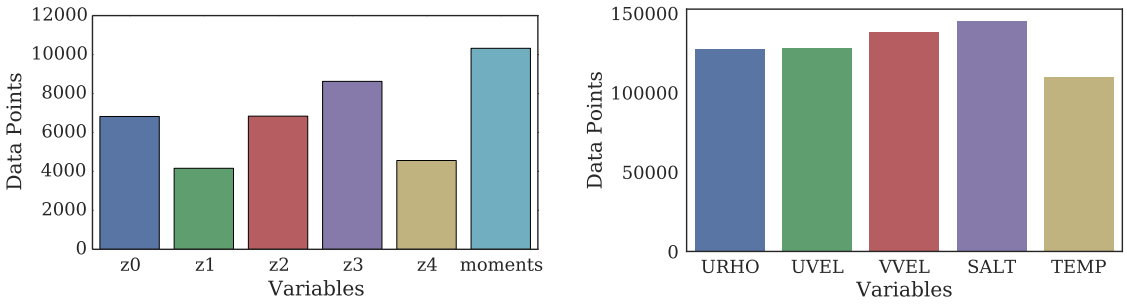
injected \ impacted	z0	z1	z2	z3	z4	z5	moments
z0	✓	✓	✓	✓	✓	✓	✓
z1	✓	✓	✓	✓	✓	✗	✓
z2	✗	✗	✓	✗	✓	✗	✓
z3	✓	✓	✓	✓	✓	✓	✓
z4	✗	✗	✗	✗	✓	✗	✓
z5	✓	✗	✗	✓	✓	✓	✓

(b) POP TAVG Output Variables

injected \ impacted	SALT	SALT2	TEMP	UVEL	VVEL
FW	✗	✓	✓	✓	✓
Gradpy	✗	✗	✓	✓	✓
RHO	✗	✗	✓	✓	✓
TFW	✗	✗	✓	✗	✓
STF	✗	✗	✓	✗	✓
SMF	✗	✗	✗	✗	✓

a method for establishing this correlation. The variables to be monitored must be carefully selected such that SDCs which occur in unmonitored variables will be captured. LADR leverages data flow analysis to identify these variables since the data-flow among crucial variables imply potential SDC propagation path.

Second, multiple data points of each contaminated crucial variable might be impacted by a single SDC, as shown in Figure 3.2. This inspired us to group data points to further reduce the overheads. As shown in our evaluation, the grouping could also improve the pre-



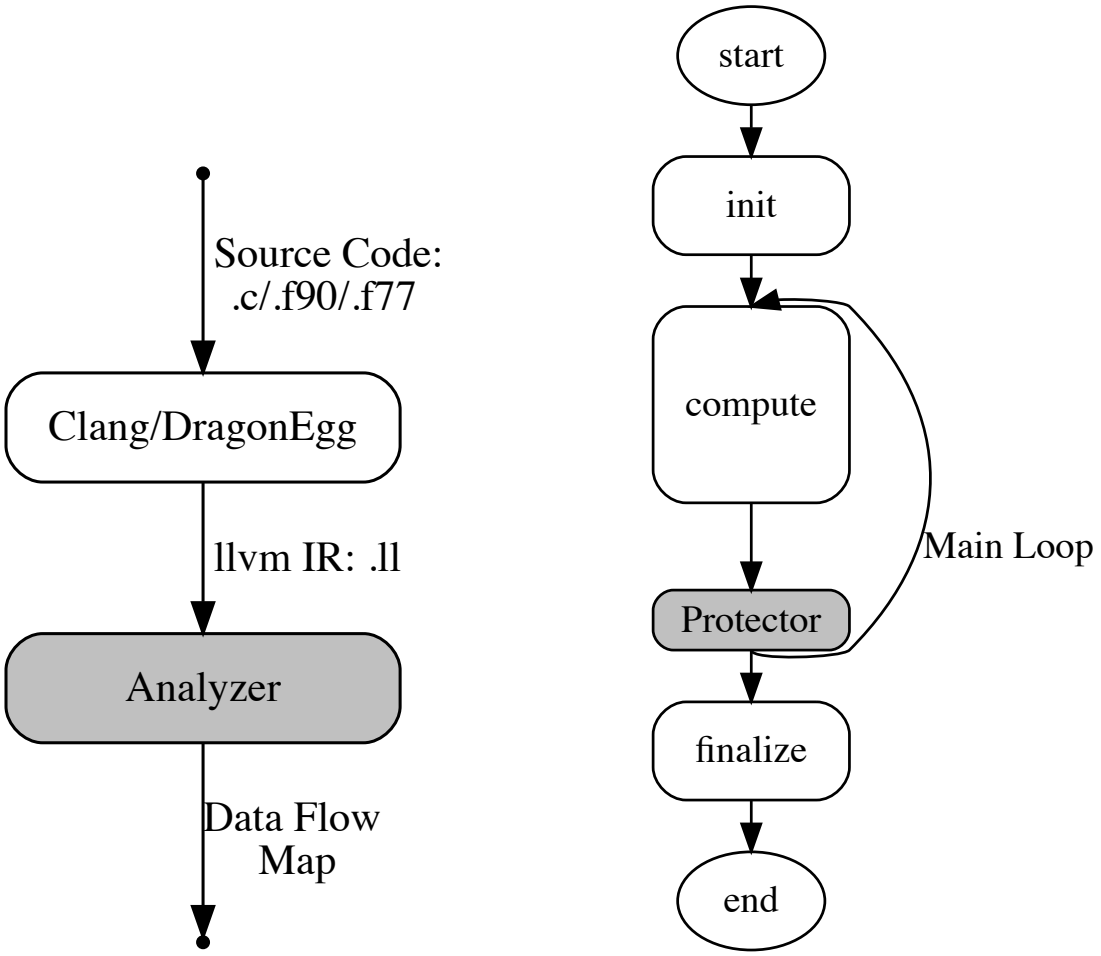
(a) GTC-P (b) POP

Figure 3.2: Average number of contaminated data points within 5 time-steps after a fault injection.

diction accuracy for the predictor in Phase I, therefore, making the detector more sensitive to SDCs.

3.3 Overview of LADR

LADR is a lightweight application-level SDC detector for iterative scientific applications. It consists of two components: 1) ***Analyzer***–a static compile-time analysis tool for identifying sink variables for a given set of crucial variables; and 2) ***Protector***–a runtime library for conducting anomaly detection. Figure 3.3 depicts the process of utilizing LADR to protect scientific applications. First, given the *crucial variables* set, the ***Analyzer*** is used to determine their sink variables. The ***Analyzer*** identifies sink variables by building a ***data-flow graph***, which depicts potential SDC propagation paths among all crucial variables. The ***data-flow graph*** is a directed graph with each node representing a crucial variable, each edge representing a potential propagation direction between two connected crucial variables, and the weight on each edge representing relative execution orders of the statement defining the propagation. Figure 3.4 gives an example of data-flow graph among 5 variables (f is an alias to e) for the code listed in Figure 3.5. Afterward, original source files are modified to apply ***Protector*** on identified sink variables. ***Protector*** is inserted at the end of the main loop of scientific applications and invoked by every MPI rank on every iteration.



(a) stage I (b) stage II

Figure 3.3: An overview of utilizing LADR

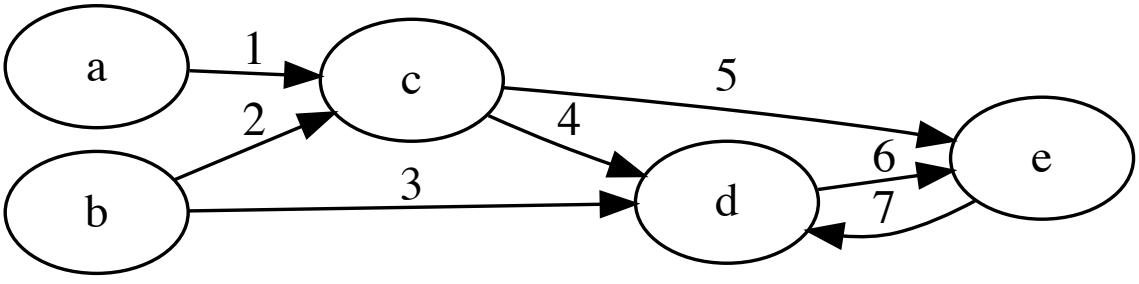


Figure 3.4: Data-flow graph for the example code in Figure 3.5

```
1  #include <stdio.h>
2  void add(double x[], double y[], double z[], int size)
3  {
4    for (int i = 0; i < size; i++)
5      z[i] = x[i] + y[i];
6  }
7
8  void mul(double x[], double y[], int factor, int size)
9  {
10   for (int i = 0; i < size; i++)
11     y[i] = x[i] * factor;
12 }
13
14 int main(int argc, char **argv)
15 {
16   double *a, *b, *c, *d, *e, *f, sum = 0;
17   int i, size;
18   a = (double *)malloc(size * 8);
19   ...
20
21   add(a, b, c, size);
22   for (i = 0; i < size; i++)
23     sum += c[i];
24   sum = sqrt(sum);
25   mul(b, d, sum, size);
26   f = e;
27   add(c, d, f, size);
28   add(f, d, d, size);
29   output(a,b,c,d,e);
30 }
```

Figure 3.5: Example code

3.4 Analyzer: Building Data-flow Graph for Identifying Sink Variables

The ***Analyzer*** extracts potential SDC propagation paths by leveraging static data-flow analysis based on the observation that SDCs would propagate from one variable – saying b, to another – saying a, only if the calculation for a directly or indirectly uses the value of b. ***Analyzer*** focuses on the main computation codes – the main loop – of scientific applications. For simplicity, the ***Analyzer*** used the following observations for scientific applications:

1. Crucial variables of these applications are life-time-long arrays (a memory space). At the program level, they are either defined as global arrays or allocated during the initialization phase of the applications but not deallocated until the end of execution.

2. Parallelization techniques (MPI and OpenMP) have no/limited impact on data-flows among crucial variables, because each process/thread conducts the same computation but on a different portion of data.[2] Hence, ***Analyzer*** ignores all MPI functions calls and OpenMP primitives.

3. Crucial variables are updated inside loops, and related loop bodies will not be skipped because of false initial loop conditions.

Analyzer works on LLVM IR code, a light-weight and low-level intermediate representation of programs. This makes LADR relatively independent of programming languages utilized by scientific applications, and it can support a majority of existing scientific applications (if not all) that are normally written in either C/C++ or Fortran. In LLVM IR code, each memory access is explicitly issued through either a load instruction (*LoadInst*)

[2]There are two principal questions WRT how MPI and OpenMP might impact LADR: 1) do they add additional dependencies for the ***Analyzer*** to track, and 2) how they impact SDC propagation. We find that for the scientific simulations we have studied, which are largely spacially decomposed, communication abstractions don't add dependencies between state variables that are not already present in the code. Second, while it is possible for SDCs to be communicated between processes, they will generally be picked up wherever they cause a significant disruption to expectations since the **protector** is embodied in every MPI rank. So these primitives get no special handling in LADR.

```
1  %2 = load i32* %i, align 4
2  %idxprom = sext i32 %2 to i64
3  %3 = load double** %x.addr, align 8
4  %arrayidx = getelementptr double* %3, i64 %idxprom
5  %4 = load double* %arrayidx,align 8
6  %5 = load i32* %i, align 4
7  %idxprom1 = sext i32 %5 to i64
8  %6 = load double** %y.addr, align 8
9  %arrayidx2 = getelementptr double* %6, i64 %idxprom1
10 %7 = load double* %arrayidx2,align 8
11 %add = fadd double %4, %7
12 %8 = load i32* %i, align 4
13 %idxprom3 = sext i32 %8 to i64
14 %9 = load double** %z.addr, align 8
15 %arrayidx4 = getelementptr double* %9, i64 %idxprom3
16 store double %add, double* %arrayidx4, align 8
```

Figure 3.6: LLVM IR Code for the loop of add

to read data from a memory location, or a store instruction (*StoreInst*) to update a memory location. Therefore, each assignment operation in source codes would correspond to several *LoadInst* instructions to read data for RHS operands, a set of related computation instructions, and one *StoreInst* instruction to update the memory with final result, as shown in Figure 3.6.

3.4.1 Extracting Data-flow among Variables

Based on the above observation, ***Analyzer*** extracts data-flow among crucial variables by analyzing *StoreInst* instructions, which have two operands named as *source* and *destination*, simply assuming that the array in *destination* (write to) covers arrays involved in *source*. For each operand of *StoreInst*, LADR leverages def-use chain to backwardly extract involved variables through looking for *LoadInst*. Based on the type of operands, there are 5 possible situations:

1. **the source operand is a pointer**, as shown in line 23 in Figure 3.5. In this case, ***Analyzer*** considers the *destination* operand as an alias to the *source* operand, and

maintains an pointer-to-pointer aliasing map for the ongoing analyzed function for future reference.

2. **the source operand is an array element and the destination variable is a scalar variable**, as shown in line 20 in Figure 3.5. In this situation, ***Analyzer*** will maintain a scalar map for the ongoing analyzed function to temporary record propagation path information among scalar variables and arrays for future reference, since it could define an indirect propagation among crucial variables.

3. **the source operand is a scalar variable and the destination variable is an array element**, as shown in line 9 ($factor$) in Figure 3.5. For this case, ***Analyzer*** will first check the scalar map using the source operand. If there exists an entry, it will retrieve related *source* arrays from the scalar map, and record the propagation information between the *destination* and each *source* array into propagation map.

4. **both source and destination operands are scalar variables**, as shown in line 13 in Figure 3.5. This is similar to the case in item 3. LADR will first check whether the *source* operand has an entry in the scalar map; if yes, it will register the destination operand into the map with the same content. Otherwise ***Analyzer*** will simply skip the instruction.

5. **both source and destination operands are array elements**, as shown in line 4 in Figure 3.5. This case defines a direct propagation among crucial variables. The ***Analyzer*** will simply register the propagation information in the propagation map (aliasing map would be checked to retrieve the actual variables before the actual registration).

3.4.2 Function Calls

To build the data-flow graph for whole application, LADR needs to handle function calls. For each *CallInst*, ***Analyzer*** takes following actions depending on the type of the callee

function:

1. **memory allocations**, e.g., malloc/alloc. For each memory allocation, ***Analyzer*** assigns an id internally to virtually represent the allocated memory region and registers it in global variable tables.

2. **basic math computation**, e.g., $sqrt$ and max. For these functions, ***Analyzer*** extracts data-flow information among their arguments and return values leveraging knowledge of library APIs.

3. **application defined functions**. For these functions, ***Analyzer*** will dive into function bodies to analyze each *StoreInst*, essentially *inlining* the subroutine to handle parameters and return values.

3.4.3 Conditional Control/Data Flow

Generally, ***Analyzer*** takes a conservative approach to conditional or run-time determined control and data flow, e.g. evaluating both paths in an *if* expression and taking the superset of those contributions as the overall data flow. This is most effectively done by simply ignoring the branch operators in the LLVM IR code.

Overall. To elaborate ***Analyzer*** clearly, we take the example code in Figure 3.5 to illustrate the entire analyzing process. ***Analyzer*** starts analysis in the main function. It considers all malloc statements as defining variables. For the first *add* function call statement (line 18), it inlines *add* by aliasing *x* to *a*, *y* to *b*, *z* to *c*, and *s* to *size*. All references to *x*, *y*, *z* and *s* are replaced with *a*, *b*, *c*, *size*, so we will get propagation paths among *c*, *a* and *b* instead of *z*, *x* and *y*. For line 20, we will get *sum* covers *c*. Since *sum* is scalar, it will be registered in scalar map. In line 21, since *sqrt* is a standard mathematical function, we can easily get that *sum* covers itself. Then for *mul* function call, it will do the inline procedure as did to *add* in above. In line 4, we will get propagation paths among *d*, *b* and *sum*. Since *sum* is now in scalar map with a propagation path from *c*, we will update the entry for *d* in

the propagation map with an edge from c. The above procedures will be repeated for the following function calls.

3.5 Protector: Anomaly-Based SDC Detection

LADR ***Protector*** shares a similar detection approach to extant anomaly-based detection methods [30, 29]. However, it distinguishes itself in two aspects: 1) it groups data points for each monitored variable, and works on the feature array constructed by extracting a data feature from each group. 2) it detects SDCs with two metrics from the perspectives of the number of contaminated data points and error magnitudes. LADR currently supports three data features for grouping, including mean, standard deviation, and entropy, as well as three predictors: Linear Curve Fit (LF), Quadratic Curve Fit (QF) (from [30]), and AutoRegression (AR). LADR seeks to select the best predictor, feature, and grouping strategy based on a learning window, such as the first few time steps of a run assuming there are no SDCs in these steps.

3.5.1 Data Points Grouping

Crucial variables of scientific applications are typically huge arrays with millions or even billions of data points, therefore, it would still incur non-negligible runtime and memory overheads with current point-wise predictions, since they need to preserve the history data and predict the value for each data point at each time-step. LADR proposes to mitigate this issue by grouping data points, which is motivated by the observation that a single SDC could contaminate multiple data points (see section 3.2). Intuitively, data points grouping could increase the potential of diluting SDCs if the group size is too large, therefore reducing fault coverage. Based on the intuition that the global/regional state could be more stable and predictable than the point-wise state (as shown in Figure 3.7), LADR employs a heuristic-based grouping algorithm to achieve a balance between fault coverage and resulting overheads. In this heuristic-based grouping algorithm, the predictability for each data

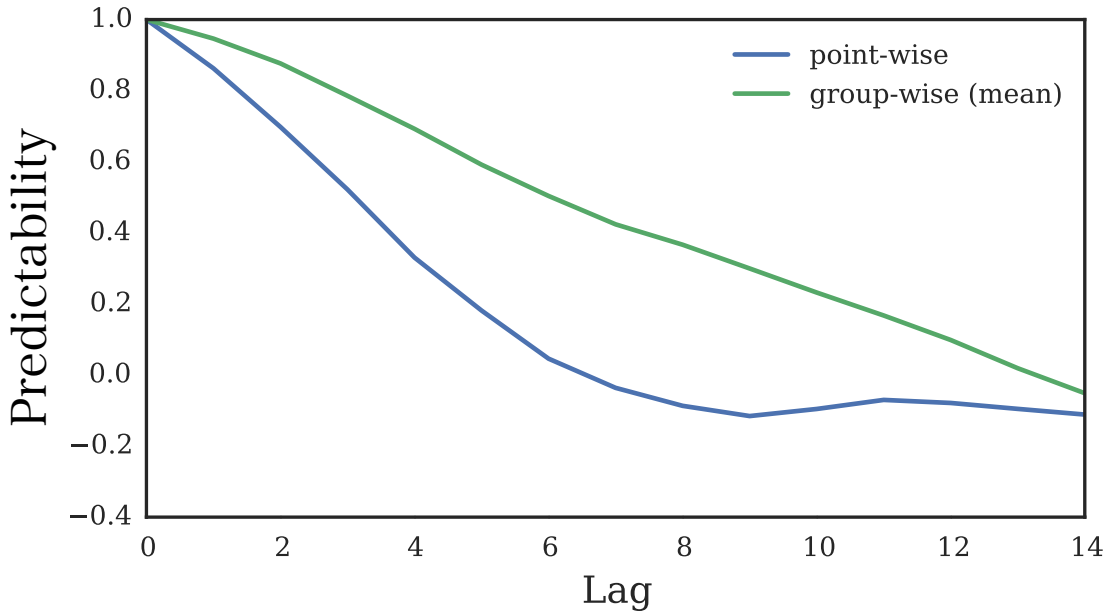


Figure 3.7: Predictability (measured with pacf) of a point and the group (constructed with 8-neighbor points) in which the point resides. The mean value of the group is used.

point(C_p) and the global array (C_g) are first calculated with partial correlation function (pacf), and data points are now roughly divided into two groups based their predictability: 1) points whose C_p are larger than C_g; and 2) points whose C_p are less than C_g. For points whose C_p are larger than C_g, they are simply divided into groups with minimum group size, which is a user-specified parameter (we set it as 8 in our experiments). For points whose C_p are smaller than C_g, they are merged with their neighbors recursively until the predictability of the group is around C_g or group size is larger than a user-specified maximal group size. For each group, the feature leading to higher predictability is selected as representative of the group. In addition, domain-specific knowledge can be leveraged for grouping too. For example, in MD codes, the velocity information (V) of each atom is encoded with three data points, representing velocities in the x, y, and z directions. Thus, a grouping strategy that calculating the absolute velocity ($\sqrt{x^2+y^2+z^2}$) can be applied, which could reduce the overhead by $3\times$. Such domain-specific grouping can replace the default heuristic-grouping algorithm or be applied together with the heuristic grouping algorithm.

3.5.2 SDC Detection

As in [30], LADR detects SDCs by checking the value of monitored variables at the end of each time-step. It works on prediction errors calculated with Eq. Equation 3.1 (below) for each data point, where P is prediction value, O is observation value and i refers to a data point. Since we have no knowledge about the correctness of observation value, historic mean values of data points are used in our estimates.

$$E(i) = (P(i) - O(i))/mean(hist(i)) \tag{3.1}$$

Afterward, for each data point, the prediction error range, $[mean(E(i) - \theta \times std(E(i))$, $mean(E(i) + \theta \times std(E(i))]$, is constructed based on its history prediction errors. θ is initialized with a constant value (we set it to 8 in our experiments), and is updated at runtime with the following equations, where γ is the actual up bound of history prediction errors:

$$\theta = \begin{cases} \frac{\theta+\gamma}{2}, & if \ \ \gamma < 0.5 * \theta \\ 1.5\gamma, & if \ \ \gamma > \theta \end{cases} \tag{3.2}$$

If the prediction error of a specific data point is out of its range, an SDC is assumed for the point. It is, however, too strict to use the result of a single point to indicate the existence of SDCs, and would incur significant false positives due to prediction variance. To mitigate this issue, LADR detects SDCs based on two metrics. First, for each time-step, we calculate the ratio of data points with out-of-bound prediction errors. An SDC is reported only when the ratio is larger than a threshold. Currently, the threshold was derived statistically from the learning window. During the learning phase for selecting best predictor, feature and grouping strategy, the out-of-bound ratio R for each time step was also recorded. And the threshold was set as $mean(R) + 5 * std(R)$, or 0.08% in our experiments. This metric is designed based on our second observation, and it is to detect SDCs that could contaminate a large number of data points. Secondly, LADR also checks the statistics (mean and standard

deviation) of prediction errors (error magnitudes across all data points) for each time-step. In normal cases, we observe that these statistics are within small bounds, which can be constructed based on the historic data using the same method for constructing prediction error bounds. If a statistic associated with the current step is out of the normal bound, an SDC is reported also. This metric could help to detect SDCs causing significant errors but affecting too few data points. The recovery procedure will be invoked if an SDC is reported by either one of these two metrics.[3]

3.6 Prototype

We implemented LADR with two separate components: a LLVM-based compiler-framework—***Analyzer***, and a runtime protector library—***GE***. In particular we implemented ***Anlyzer*** as an independent LLVM pass ($\sim$ 1688 LOC) based on LLVM-3.5.2. It analyzes unmodified source code of an application to build data-flow graph among crucial variables. We leverage Clang or DragonEgg to compile C/C++ or Fortran codes into LLVM IR codes (.ll file). Our current prototype of ***Anlyzer*** only generates the data-flow graph, and the sink variables are manually/visually selected by simply picking destination node in the generated graph, e.g., d in Figure 3.4. We hope to fully automate the entire process in future work.

The GE runtime is developed based on GSL library. Similar to [30], it exposes users 5 API routines as shown in Figure 3.8. *GE_init* is inserted to the beginning of the application right after *MPI_Init*. *GE_Protect* is inserted before the main loop for each protected sink variable. Each sink variable can be assigned with separate parameters according their data features. *GE_Snapshot* is inserted at the end of main loop, right before output routine. Finally, *GE_PrintResult* and *GE_Finalize* are expected to be inserted right before *MPI_Finalize*.

[3]This paper doesn't propose new recovery methods, and we assume checkpoint/restart can be used here.

```
1 GE_Init(MPI_COMM);
2 GE_Protect(char varname, void *var, int data_type, ↩
    size_t size, int buf_size, float ratio, float ↩
    impaction);
3 GE_Snapshot();
4 GE_PrintResult();
5 GE_Finalize();
```

Figure 3.8: GE Protector API

3.7 Evaluation

We evaluated LADR through fault-injection experiments. In this section, we will first introduce the evaluation methodology, and then present evaluation results.

3.7.1 Evaluation Methodology

We evaluated LADR on a cluster with 32 nodes. Each node is equipped with a 12-core Intel(R) Xeon(R) X5660 (2.80GHz) CPU, 24GB of memory and Mellanox Technologies MT26438 InfiniBand card. Similar to the work in [30], we focused on unexpected data changes (SDCs) caused by transient faults, e.g., bit-flips of the data. We simulated SDCs through application-level fault injections as did in study [19]. We implemented a simple fault injection tool based on GDB's MI interface for this purpose. For sake of easy analysis, the tool injects faults to one MPI rank during each run of applications. In this tool, a fault is identified by a tuple with 4 elements (***iteration***, ***execution point***, ***target***, ***fault***):

- ***iteration*** and ***execution point*** together determine when the fault will be injected.
- ***Execution point*** is represented in form of *file:line*.
- ***target*** determines where the fault will be injected. It is a specific memory location of applications.
- ***fault*** determines how to corrupt the value of the target. Random bit-flips are used in our evaluation.

We injected faults directly into applications' memory space since it behaves closer to SDCs. We performed one injection per run, and contaminated one data point per injection. $\sim$6000 injections in total were performed. We compared LADR to the "Reference" scheme, in which all of crucial variables in our setups were monitored and point-wise predictor was employed. In contrast, LADR only monitored the identified sink variables (shown in Table 3.4) and also applied the data point grouping technique. For the "Reference" scheme, the tool developed in [30] was used. We compared them from 3 aspects:

1. *fault coverage (FC)*. It measures how many SDCs are detected by an SDC detector. It is defined by number of detected SDCs over the total injections that contaminated values of crucial variables.

2. *false positive (FP)*. A FP happens when an SDC is mistakenly reported for a fault-free time-step. It is defined as the number of mistakenly reported time-steps over the total number of iterations under the evaluation.

3. *overhead*. It contains runtime overhead incurred by predictions, and memory overheads incurred for storing history data sets.

3.7.2 Evaluated Workloads and Baselines

We evaluated LADR with the four scientific workloads in Table 3.2. In our evaluation, we set up the baselines for GTC-P and POP with 6 crucial variables, and for miniMD and LAMMPS with 3 crucial variables. The data size of each crucial variable is shown in Table 3.3. LADR's ***Analyzer*** identified 1 sink variable for each evaluated workload, as shown in Table 3.4. After sink variables were determined, application codes were modified to monitor the variable using the GE library. The modification effort involved 25 lines of code changes for GTC-P, miniMD and LAMMPS, as well as 45 lines of code changes for POP, which was minor and acceptable.

Table 3.2: Evaluated scientific workloads

Workloads	Descriptions	crucial variables
Gyrokinetic Toroidal Code–Princeton (GTC-P)	C program. A 2D domain decomposition version of the GTC global gyrokinetic PIC code for studying micro-turbulent core transport. It solves the global, nonlinear gyrokinetic equation using the particle-in-cell method.	6 particle data variables: zion0, zion1, zion2, zion3, zion4, zion5
Parallel Ocean Program (POP)	Fortran program. A 3D ocean circulation model designed primarily for studying the ocean climate system. It was used to perform high resolution global ocean simulations to resolve meso-scale eddies that play an important role in the dynamics of the ocean.	6 TAVG output variables: uvel, vvel, salt, temp, rho, salt2
LAMMPS	C++ program. a classical molecular dynamics code, and an acronym for Large-scale Atomic/Molecular Massively Parallel Simulator developed at Sandia National Laboratories	3 molecular property variables: X, F, V
MiniMD	C++ program. A simple, parallel molecular dynamics (MD) code developed at Sandia National Laboratories	3 molecular property variables: X, F, V

Table 3.3: Per time-step size of evaluated variables

GTC-P		POP		miniMD		LAMMPS	
name	size	name	size	name	size	name	size
zion0	25280 KB	TEMP	5750 KB	V	16384 KB	V	16384 KB
zion1	25280 KB	SALT	5750 KB	F	16384 KB	F	16384 KB
zion2	25280 KB	SALT2	5750 KB	X	16384 KB	X	16384 KB
zion3	25280 KB	UVEL	5750 KB	–	–	–	–
zion4	25280 KB	VVEL	5750 KB	–	–	–	–
zion5	25280 KB	RHO	5750 KB	–	–	–	–

Table 3.4: Analyzer cost

Apps	Source code (LOC)	LLVM IR (LOC)	Sink variables	Analysis time
GTC-P	18,453	61,049	moments(1775KB)	3 mins.
miniMD	4,167	28,028	V(16384 KB)	2 mins.
LAMMPS	415,084	1,655,405	V(16384 KB)	18 mins.
POP	59,678	478,311	TRACER(575KB)	189 mins.

3.7.3 LADR Analyzer Cost

As shown in Table 3.4, ***Analyzer*** worked more efficiently on GTC-P, miniMD and LAMMPS than on POP. It roughly took $2 \sim 3$ minutes for analyzing GTC-P and miniMD, ~ 18 minutes for analyzing LAMMPS, but nearly 3 hours for POP. We attribute this issue to two reasons: 1) a larger code base for POP (around $3\times$ of GTC-P and $10\times$ of miniMD), and 2) inefficient IR code generation for Fortran. During our evaluation, We found that the IR code generated from the Fortran program was not as succinct as the IR generated from C/C++. It introduced significantly more branches and virtual functions, which are not shown in the original source code. These features, especially branches, complicate the analysis of ***Analyzer*** because it needs to evaluate each potential execution path. With the ongoing project FLANG, a native Fortran front-end for the LLVM framework, we expect this issue would be mitigated.

3.7.4 Fault Coverage

Fault coverage is a major metric for measuring the effectiveness of SDC detectors. While LADR aims to optimize the runtime and memory overheads, it should not significantly sacrifice fault coverage.

Figure 3.9 compares the fault coverage of LADR to the "Reference" scheme for the evaluated workloads. Results of using different grouping strategies are also reported. "LADR-grouping(H)" shows grouping data points leveraging the proposed heuristic grouping algorithm, "LADR-grouping(S)" divided data points evenly using the same number of groups as in "LADR-grouping(H)," and the "LADR" label shows simple point-wise mon-

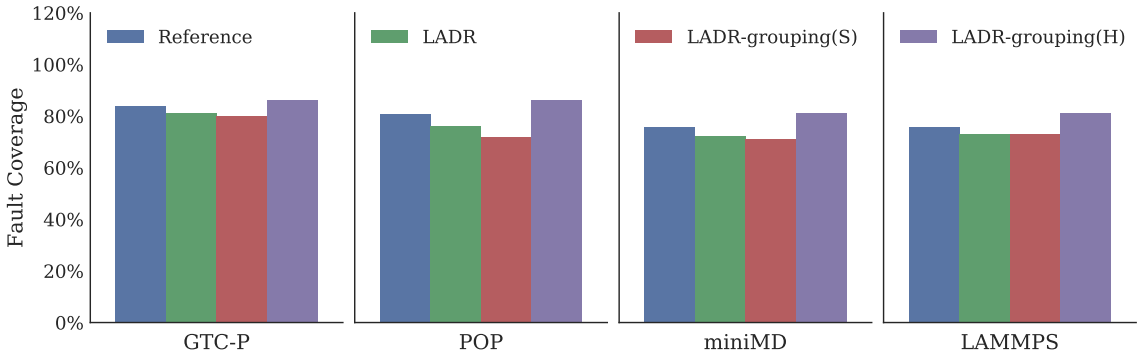
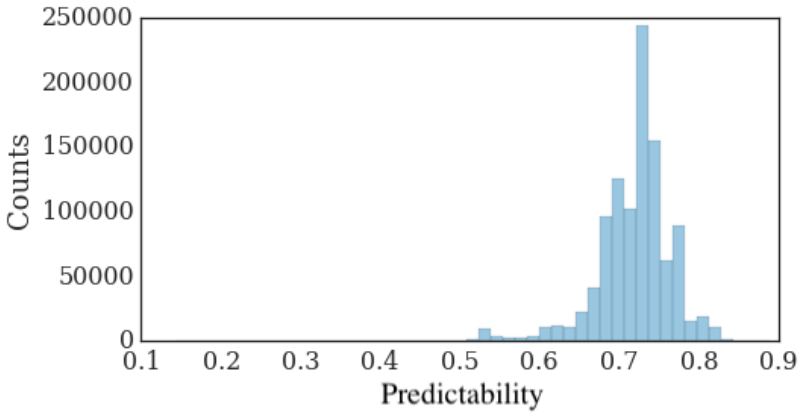


Figure 3.9: Fault coverage comparison. H – **H**euristic grouping; S – **S**tatic grouping

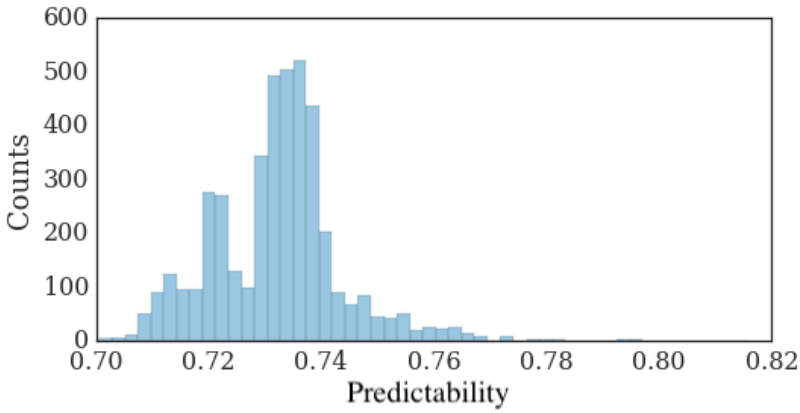
itoring of the sink variable. As shown in the figure, "LADR" achieved comparable fault coverage as compared to the "Reference" scheme, reducing overhead by monitoring fewer variables at a cost of just a $1\% \sim 4\%$ decrease in fault coverage. Meanwhile, the heuristic grouping algorithm boosts its performance significantly, since it improved the predictability of the feature data, therefore allowing a more accurate detection model. As an example, Figure 3.10 presents the impact of our heuristic grouping algorithm on the predictability for miniMD. It shows that, the heuristic grouping algorithm effectively removes the data points with less predictability (comparing Figure 3.10(a) and Figure 3.10(b)), but didn't blindly increase the group size Figure 3.10(c). As shown in Figure 3.1(b), low predictability would lead to lower fault coverage since a large bound was required for tolerating false positives. These results suggest that, by leveraging data-flow information, it's unnecessary to monitor all crucial variables to protect scientific applications from SDCs, and heuristic grouping algorithm can achieve a better balance between group size and prediction accuracy than static grouping.

3.7.5 Runtime and Memory Overheads

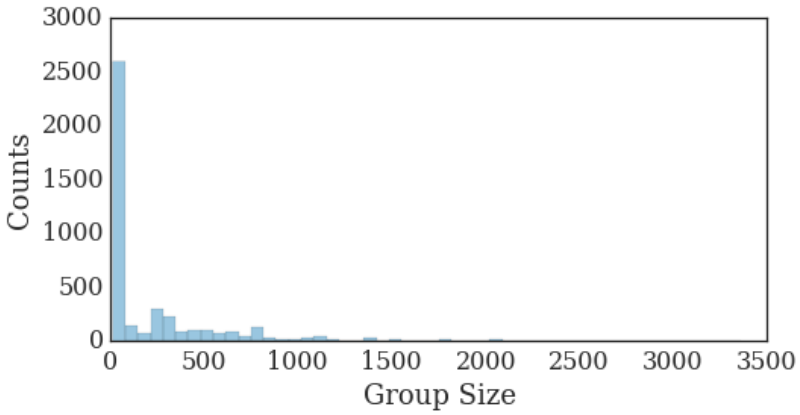
In this section, we evaluated the overheads of LADR. We compared it to the "Reference" scheme and to baseline runs in which no protection is applied to the workloads. Figure 3.11 presents their overheads normalized to the baseline for each evaluated workload. By moni-



(a) predictability (point-wise)

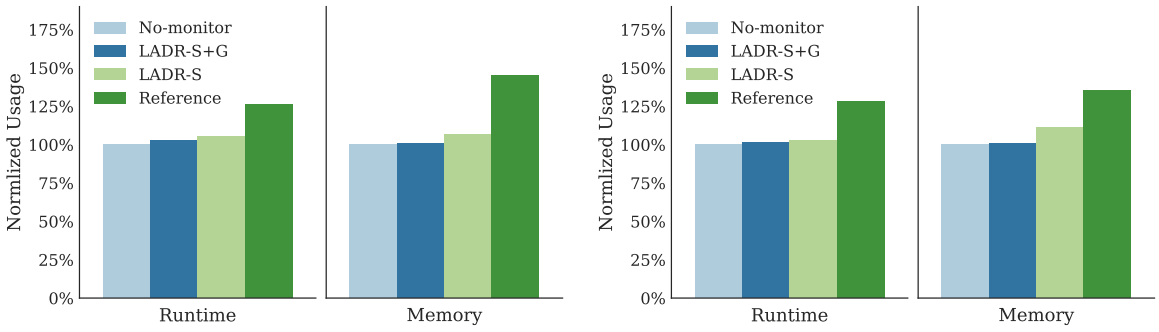


(b) predictability (group-wise)

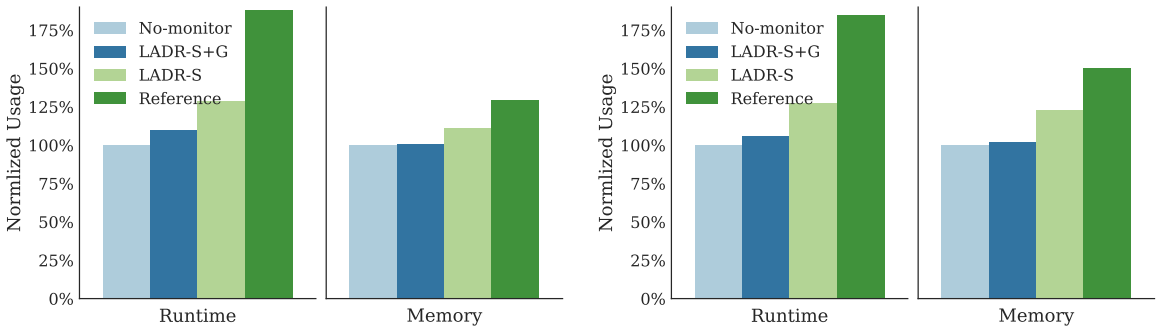


(c) group size distribution

Figure 3.10: Impacts of heuristic grouping. The constructed feature data is more predictable among time-steps. Most groups have a few data points, while static grouping has 245 points for each group. Due to limited space, only the data for miniMD is plotted. Other workloads share a similar observation.



(a) GTC-P (b) POP



(c) miniMD (d) LAMMPS

Figure 3.11: Runtime and memory overheads of LADR with sink variable (**S**) and data points grouping (**G**)

toring sink variables, LADR reduced runtime overheads by 21% for GTC-P, 25% for POP, 22% for miniMD and 57% for LAMMPS. It also reduced memory overheads for them respectively by 38% for GTC-P, 39% for POP, 24% for miniMD and 28% for LAMMPS. This is mainly because it monitored fewer variables. For GTC-P and POP, it is also because the selected sink variable is smaller than the crucial variables. In addition, the grouping algorithm divided GTC-P into 5230 groups, POP into 3270 groups, miniMD into 4289 groups, and LAMMPS into 10367 groups for data points in each node. This further reduced memory overheads to around 1% for these workloads and runtime overheads to 2.96% for GTC-P, 1.22% for POP, and 9.16% for miniMD, and 11.8% for LAMMPS. These results show that monitoring sink variables can significantly reduce overheads of anomaly-based SDC detectors, and data points grouping can further reduce the overheads to a negligible level.

In conclusion, the evaluation results show that LADR significantly reduced runtime and memory overheads as compared with prior methods without sacrificing fault coverage. It would be a promising method for scientific applications that can tolerate some numerical disturbance while being protected from larger SDCs in the majority of their data.

3.8 Related Work

In addition to aforementioned data-anomaly based approaches, current researches against SDCs mainly fall into two categories: 1). computing replication and 2). algorithm-based fault tolerance (ABFT).

First, computing replication performs SDC detection and correction by either duplicating (critical) instructions of applications via the compiler technique [28, 39, 27, 40] or replicating the execution of the same copy of application on different sets of computing nodes [4, 26, 25]. Thus multiple instances of the computation (instruction or process) can be compared and divergences are taken to be evidence of an SDC. EDDI [27] and SWIFT [28] are two important representatives for instruction-level redundancy. EDDI duplicated all instructions and inserted "checking" instructions right before storing a register

value back to memory or determining the branch direction. SWIFT optimized overheads of EDDI through an enhanced control flow mechanism. Regarding to the application-level redundancy, RedMPI [25] ran a shadow process for each MPI rank and redesigned MPI communication system to duplicate message passing for both shadow process and principle process. SDCs were detected through comparing messages between replicas. ACR [4] replicated processes on different nodes and detected SDCs by comparing check-pointing data from each replica.

On the other hand, ABFT techniques [41, 42, 43, 44] mainly focus on designing resilient data structures and algorithms for processing matrix. Chen et al. [41] examined the block row data partitioning scheme for sparse matrices, which were then utilized to recover critical data without checkpoint. Du et al. [42] constructed a column/row checksum matrix for matrix computations, such that SDCs can be detected by scanning partial product matrix and recovered with the checksum matrix. These algorithm-specific methods are highly specialized and as such focus on specific computational kernels, which are normally a small part of scientific applications.

3.9 Conclusion

In this chapter, we presented and evaluated LADR, a lightweight tool to help validating a given simulation run to be free of SDCs. In the absence of such validation, the integrity of high fidelity simulations remains questionable on very large scale systems that are prone to such errors. An effective solution must achieve high fault coverage with limited overheads. To this end, LADR is built on top of state-of-the-art data-anomaly detection techniques, and extended them with program analysis to minimizing runtime and memory overheads. In particular, it exploits the correlation among state variables and data points, such that it can detect SDCs by only monitoring the data in a small portion of process's memory. We evaluated LADR using application-level fault injection experiments. Results suggest that LADR can protect application from influential SDCs with no more than 8% overheads. For

two of evaluated workloads, it only incurs around 2% overheads. LADR demonstrates that it is unnecessary to apply anomaly detection techniques on all crucial variables. Instead, a subset of crucial variables can be identified employing compile-time data-flow analysis.

CHAPTER 4

COMPILER-ASSISTED RECOVERY FROM SOFT FAILURES

This chapter presents a light-weight compiler-assisted recovery framework for soft failures. The framework aims to repair the (crashed) process on-the-fly when a crash-causing error is detected, such that applications can continue their executions instead of being simply terminated and restarted. The framework consists of a compiler component, which performs necessary code transforms and construct a recovery kernel for each crash-prone instruction during the compilation of applications, and a runtime system which attempts to repair corrupted state of the process upon an occurrence of an error by searching and executing the constructed recovery kernel to recompute the corrupted architectural state on-the-fly for the impacted process. The framework also exploits side effects introduced by code optimization techniques such as strength reduction and loop unrolling, which are widely adopted by modern compilers, to improve its fault coverage against transient hardware faults.

4.1 Introduction

As discussed in chapter 1, for scientific applications, soft failure is one of two major outcomes of transient hardware faults. However, while there has been significant amount of prior work on detecting and correcting SDCs [41, 43, 37], less research effort has gone into handling soft failures, perhaps because the community takes it for granted that the standard Checkpoint/Restart (C/R) methods in current resilience mechanisms can provide adequate recovery. Unfortunately, while the C/R technique does have the capability to recover from soft failures, it is very costly in terms of lost opportunities (batch job slots), lost computation (everything since the last checkpoint) and I/O overheads (repeatedly writing checkpoint files). These costs are particularly significant for massively parallel jobs [3, 5, 45]. Reliance on C/R means that a job which suffers a transient hardware fault will be killed

and must be resubmitted, having to wait in the job queue before it can continue execution. Once running, the job must first load the saved state from a checkpoint, which would involve slow I/O operations and then redo calculation that was lost before it can get to the point where the failure occurred. However, since transient faults only impact the hardware temporarily, i.e., for a few cycles, more advanced techniques might mask the fault at the application level, allowing the application to continue the normal operations when their corrupted state is repaired, therefore improve the system utilization efficiency and speedup the process of modern scientific discoveries.

In this chapter, we argue that, while the C/R is still necessary in many circumstances (e.g., power failures, job termination, etc.), soft failures can often be handled with a lightweight resiliency mechanism, which could help to mitigate the overall overheads of the resilience mechanism in HPC systems. Specifically, we propose **CARE** (**C**ompiler-**A**ssisted **RE**covery), a lightweight and compiler-assisted framework to recover processes of scientific applications from soft failures on-the-fly. The framework is designed for targeting errors emanating from computing logic units, assuming that memory areas are protected with ECC. Upon a failure, the framework will attempt to diagnose and repair the corrupted state for the failing process on-the-fly through replaying related computations, such that scientific applications can continue their executions instead of being simply terminated.

CARE is inspired by two insights we observed from scientific applications and our study about the manifestation of soft failures (See section 4.2 and section 4.3):

1. The majority of soft failures would manifest via hardware traps. Specially, as much as 98.95% (91.45% on average) of soft failures evidence themselves by causing a *SIGSEGV* because of invalid memory access. This is because many scientific applications contain features like stencil codes, or otherwise involve complex address computations to access neighbor values.

2. Most soft failures would manifest within a few dynamic instructions. Hence, the original raw data used for array location computations remains uncontaminated, and

```
1 // memory access statement
2 phitmp[(mzeta + 1) * k] = mzeta * k;
3
4 // the recovery kernel
5 uint64_t recovery_kernel(int *phitmp, int mzeta, int k)↩
      {
6     return (uint64_t)(phitmp + ( mzeta + 1 ) * k);
7 }
```

Figure 4.1: A sample recovery kernel.

can be used to recompute the corrupted state.

Based on the above insights and the predominance of soft failures manifesting as invalid memory accesses, the approach employed by **CARE** builds a set of *recovery kernels*, one per memory access instruction, that can recompute appropriate addresses for failed dereferences at runtime.

A recovery kernel simply consists of a standard function which mirrors the address calculation operations of a portion of the application. An example of *recovery kernel* is shown in Figure 4.1. This kernel computes the address for $phitmp[(mzeta + 1) * k]$, taking $phitmp$, $mzeta$ and k as parameters. Upon a *SIGSEGV* failure raised when accessing $phitmp[(mzeta+1)*k]$, **CARE**'s runtime system will fetch values of $phitmp$, $mzeta$ and k from the address space of the process, and execute the kernel to recompute the address. **CARE** can successfully get the correct address if $phitmp$, $mzeta$ and k are unmodified by the fault. Otherwise, **CARE** will get a invalid address as issued by the failed instruction and will terminate the process. To this end, a challenge for **CARE** is corruptions in loop induction variables, which are commonly involved in address computations for array element accesses and are partial of parameters of recovery kernels. Once their values are corrupted by transient hardware faults, it could significantly reduce the recovery rate of the framework for applications. To improve its recovery capability in cases when loop (derived-)induction variables are corrupted by the fault, **CARE** will exploit side effects introduced by code optimization techniques such as strength reduction and loop unrolling,

which are widely adopted by modern compilers, to repair these corruptions. While these code optimization techniques were mainly designed to improve the execution speed, they introduce equivalent computation patterns and values (semi-redundancies) into the code, providing opportunities which can be exploited for recovery of corrupted induction variables. However, these semi-redundancies are typically not explicitly present. To this end, **CARE** introduces two extra code transformations to reshape the code without introducing significant performance penalties, such that the introduced semi-redundancies can be explicitly exposed to the underlying recovery method.

In summary, this chapter makes the following contributions:

- We propose a new failure recovery framework for scientific applications to survive soft failures. It exploits hardware detection of memory access violations to repair crashed architecture states on-the-fly by replaying computations that are extracted from applications. It is lightweight. Except requiring some offline code analysis effort for building recovery kernels, it incurs almost **zero** runtime overhead and fixed 27MB memory overheads during the normal execution of applications.

- To motivate the design of the new framework, we studied the manifestation of soft failures in modern scientific applications through empirical instruction-level fault injection experiments. We classified the soft failures based on hardware trap symptoms, and examined their manifestation latency measured in terms of number of dynamic instructions. The results of this empirical study motivated the design of the new framework.

- We described how to exploit the properties of modern code optimization techniques, coupled with two extra code transformations, for building a lightweight (if not zero-overhead) failure recovery method. To the best of our knowledge, this is the first work to examine how code optimization techniques can contribute to lightweight resilience mechanisms.

- We designed and implemented the proposed strategy based on the LLVM framework and the Linux system. While more engineering work is needed to support -O2/-O3 optimizations, our prototype demonstrates a solid step towards a lightweight resilience mechanism for soft failures.

- We evaluated the proposed framework with 4 scientific workloads and with up to 3072 cores. The results show that, on average, it can recover about 84% of soft failures for the evaluated workloads within dozens of milliseconds, allowing parallel applications to finish their jobs with almost no delays even when crash-causing errors happen during their execution. We also present preliminary evaluation results for *BLAS*, showing that the proposed method can support for failure recoveries in libraries with a high coverage rate and negligible performance hit.

The rest of the chapter is organized as follows: section 4.2 studies and presents how soft failures are manifested from transient faults, which motivated the design of **CARE**; section 4.3 explains why **CARE** is important for many scientific applications, and the challenges it needs to address; section 4.4 briefly introduces code optimization techniques exploited by **CARE**; section 4.5 depicts the overall picture of the system; section 4.6, section 4.7 and section 4.8 respectively present design details about the front-end and the runtime of **CARE**; section 4.9 presents the prototype details; Next, evaluation results are presented in section 4.10, and the related state-of-the-art studies are discussed in section 4.11. Finally, we present our conclusion in Section section 4.12.

4.2 Manifestation of Soft Failures

In this section, we will study how are soft failures typically manifested from transient faults, and present the insights that motivated the design of **CARE**.

With increasing concerns about transient faults from HPC communities, a solid understanding about the manifestation and propagation of transient faults is key to building an

efficient resiliency mechanism. Several recent papers [18, 19, 21] have studied the impact of transient faults on scientific applications leveraging empirical fault injection experiments. While all of these studies point out a need for an efficient application level resilience mechanism against soft failures for scientific applications running on future extreme-scale systems and some, such as [46], examine the propagation of SDCs, none provided quite the insights necessary for devising efficient mechanisms for fault recovery. In their studies, they treat applications as black-boxes. In particular, they do not provide adequate information about how soft failures manifest and propagate inside applications, which is critical for building application level resiliency mechanisms. Here, we focus on exploring how transient faults manifest, propagate and lead to soft failures (crashes). We are specially interested in: 1). determining the major causes/symptoms of soft failures; and 2). the latency of their manifestation in terms of number of instructions executed from the injection point to the crash point. We built a new instruction-level fault injection tool which allows us to track the propagation of faults from instruction to instruction. Our method first injects faults into target operands of randomly selected dynamic instructions. The faults are then allowed to propagate while a trace is captured and analysed. We performed empirical fault injection experiments on five representative scientific workloads, including HPCCG, CoMD, miniMD, miniFE, and GTC-P (described in Table 4.7), and analyzed the injections that led to soft failures to find common patterns that can be exploited by a recovery mechanism. These workloads are from different scientific domains e.g., plasma physics, molecular dynamics, etc., and implementing different algorithms, such as Lennard-Jones potential, embedded atom model, and conjugate gradient. For each workload, we performed 10 000 injections based on the single-bit-flip fault model. In the rest of the section, we will detail the methodology of experiments, and the insights we gleaned from this study.

4.2.1 Methodology

We build the fault injection tool with GDB and Python. The tool at runtime attaches itself to the target process randomly, and then injects a fault to the "destination" operand of the instruction at the attachment point. A "destination" operand is one of architecture states, e.g. a register, or a memory cell, that is updated by the instruction. We simulate transient faults from the CPU logic by randomly flipping a bit of the value in the "destination" operand[1]. As done in previous studies [21, 20], we chose a single-bit-flip fault model since it is a conservative way to estimate the causes and the latency for soft failures, considering that multi-bit-flip faults are more likely to incur soft failures with lower latency than single-bit-flip faults [19]. We utilized capstone [47], an instruction disassembly framework, to disassemble the instruction and get its semantic information for identifying destination operands. The fault is injected at the point right after the instruction is executed, then execution is continued, tracking fault propagation by recording its execution path. For each run of an application, only one injection is performed. The trace of instructions that propagate the fault is then analyzed.

4.2.2 Results and Insights

We categorized the general outcomes of injections into 4 groups: Benign, Soft Failure, SDC, and Hang. A transient fault is benign (or in short vanishes without causing any change in execution) if it doesn't have impact on the application. In such cases, the faulty value could either refer to an incorrect but valid memory location containing the same value to the original memory location, or its effect is masked by a program operation (e.g., min/max operator that masks injections max/min operand, or bit-wise logical operation that suppresses most or least significant bits). Otherwise, it will either kill a process (Soft Failure), lead to incorrect outputs (SDC), or result in a hang state where there is no progress

[1]Where the destination operand is implicit, e.g. X86 *idiv %ecx* which divides the value in $\%edx : \%ead$ by $\%ecx$ and store results in $\%eax$ and reminder in $\%edx$, one of the implied destinations, e,g, $\%eax$, is selected.

Table 4.1: The overall outcomes of fault injections

Workloads	Benign	Soft Failure	SDC	Hang
HPCCG	3118	3409	3472	0
CoMD	6433	2439	1120	8
miniFE	5073	3518	1376	9
miniMD	951	4065	4984	0
GTC-P	6875	1644	1479	2

Table 4.2: Breakdown of soft failures based on symptoms

	SIGSEGV	SIGBUS	SIGABRT	Other
HPCCG	3322	32	22	33
CoMD	2195	57	41	146
miniFE	3447	51	6	35
miniMD	4028	6	25	6
GTC-P	1196	49	375	24

on execution. As presented in Table 4.1, even though majority of faults are benign, around 30.15% of them manifest as soft failures, and 24.86% of them lead to SDCs. While faults happening in FPU are more likely to cause SDCs, the faults manifested in ALU instructions are more likely to lead to soft failures. There has been a significant amount of work on detecting SDCs but soft failures that cause application crashes have received less attention. Once an application crashes, it needs to be restarted incurring costly recovery operations using check-pointed values.

Table 4.2 breakdowns the soft failures based on symptoms. It shows that, most ($72.75\% \sim 98.95\%$, 91.45% on average) of soft failures manifest as *SIGSEGV*, typically because they corrupt address calculations and lead applications to access invalid memory locations. In addition, Table 4.3 presents the latency distribution for single-bit-flip model. As it shows, the vast majority of soft failures ($> 83\%$) were manifest within 50 or less dynamic instructions, with more than half of them manifesting within 10 dynamic instructions. We

Table 4.3: Latency distribution for soft failures

	Latency (Instructions)			
	≤ 10	$11 \sim 50$	$51 \sim 400$	> 400
HPCCG	99.09%	0.482%	0.602%	0.301%
CoMD	64.15%	23.57%	7.43%	4.85%
miniFE	48.03%	37.15%	12.4%	2.407%
miniMD	53.65%	22.09%	0.03%	24.23%
GTC-P	52.68%	28.76%	9.7%	8.86%

```
1 for (i = ipsi_in1; i < ipsi_out1+1; i++){
2  for (k = 0; k < mzeta+1; k++) {
3   phitmp[(mzeta + 1) * (igrid[i] - igrid_in) + k] =
4     phitmp[(mzeta + 1) * (igrid[i] + mtheta[i] - ↩
        igrid_in) + k];
5  }
6 }
```

Figure 4.2: Stencil Code Structure

believe such low-latency manifestation implies that the original values (stored in registers or memory) which were involved in the address computation were likely to be intact during this latency window, and that it might be possible to recover the calculation and essentially mask the fault by creating mechanisms to access these original values to recompute the effective address (which is destroyed due to the fault). Based on these two insights, the proposed framework is designed specially to protect memory access instructions. During the compilation of applications, it will build a recovery kernel for each memory access instruction by cloning its address computations. This kernel will then be utilized to recompute the address when the instruction is contaminated by a fault.

4.3 Why CARE is Important to Scientific Applications ?

Modern scientific applications typically contain stencil codes, which are a class of iterative kernels. They employ huge arrays to store computation states. The core computation of

these applications is to update elements of this arrays according to some fixed pattern using neighboring array elements. Such stencil pattern of data access is repeated for each element of the array. On one hand, due to this access pattern, applications have to maintain several data structures (mainly arrays) for storing the neighbor information. Therefore, to update an element in scientific data arrays, some amount of address calculation is required to access neighbor cells. **CARE is directly motivated by such complex address calculations that exist inside these scientific applications**. As an example, Figure 4.2 presents a piece of code extracted from GTC-P. It demonstrates that the code involves non-trivial address calculations when accessing an array element in $phitmp$, including 3 or 4 additions, 1 subtraction, and 1 multiplication. For many scientific workloads, as shown in Table 4.4, there exhibit some large percentage of memory accesses (generally $86.85\% \sim 94.08\%$) having multiple binary operations in their address calculations; and each memory access instruction, on average, would involve $2.96 \sim 5.6$ binary operations. In addition, inside these applications, some variables used in the address calculation are infrequently updated during their executions, once every few time steps or even unchanged during the whole execution. Consider the GTC-P code in Figure 4.2 for example, $igrid$, $mtheta$ are never updated after they are initialized, and $igrid\_in$, $mzeta$ are unchanged when executing the loop. Because of the complexity of the address computations and the infrequently updated raw data for the computations, the invalid-memory-access errors due to transient faults are recoverable with a high probability for these situations. For example, if a failure manifests when accessing $phitmp[(mzeta + 1) * (igrid[i] - igrid\_in) + k]$, the fault could have happened when updating i, k or the rest of computation, such as $mzeta+1$. While the prior cases are likely unrecoverable, the later cases are definitely recoverable. As comparison, a failure which occurs when accessing $phitmp[i]$ is less likely recoverable since the failure would imply a corrupted value of i, and the likelihood of the correct original value for that still existing is potentially quite small. Said another way, a failure manifesting in an address calculation which involves a large number of temporaries is naturally more likely

Table 4.4: The percentage of memory access instructions involving multiple computations in their address calculations, and average number of involved operations

	HPCCG	CoMD	miniFE	miniMD	GTC-P
No. Insts	91.49%	94.05%	94.08%	89.22%	86.85%
Avg. No. ops	4.62	5.6	3.04	2.96	3.60

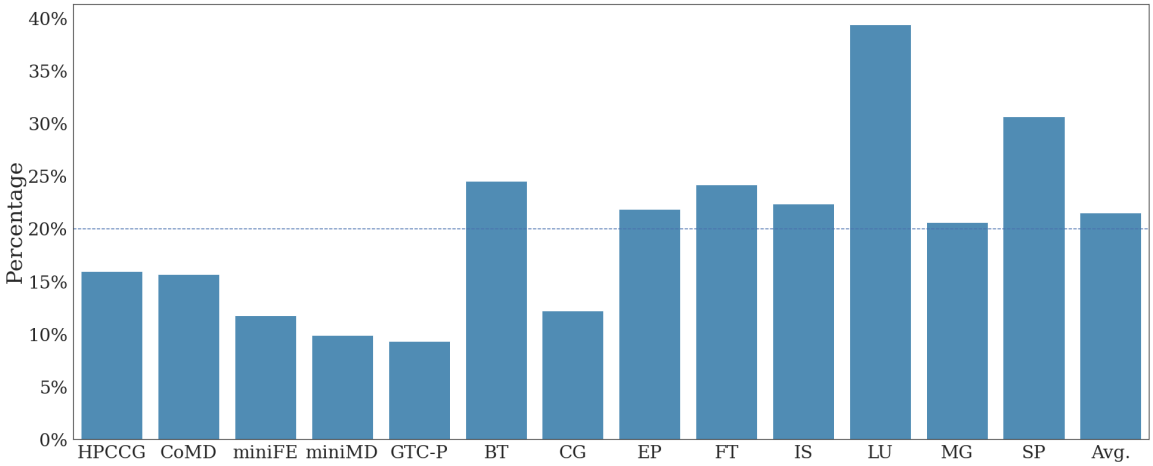


Figure 4.3: The portion of computations (estimated) dedicated to updating induction variables

to be recoverable than one which does not.

On the other hand, the core computation for updating array elements are typically expressed as loops, and array subscript calculations based on the induction variables are essential components for retrieving the array elements. As discussed before, once induction variables are corrupted, the array subscript calculations get corrupted and processes may crash due to accesses to nonexistent array elements outside their bounds. Unfortunately and perhaps surprisingly, updating loop inductions variable could contribute significant portion of computations in many of these applications, thus they stand a good chance of being corrupted by transient faults in ways that are very challenge to be recovered and need special attentions. Figure 4.3 roughly estimates the portion of induction-variable computations for several benchmarks (see Table 4.7 for details). We statically counted the number of instructions that are involved in updating induction variables based on LLVM IR representations of applications. They were normalized to the total number of instructions inside loops of

these applications [2]. It surprisingly shows that, for these benchmarks, around $9\% \sim 40\%$ (average 22% in our applications) of computation is involved in induction variable updates. If this seems unintuitively large, (e.g. almost 40% of computation) recall that code optimization tends to convert regular patterns of array accesses into pointer-based induction variables and such accesses could dominate inner loops in these applications. Transient hardware faults striking these instructions by their nature tend to corrupt address computations, lead the process to access incorrect array elements (leading to incorrect outputs) or invalid memory addresses (leading to process crashes). Sharma et al. [48] show that, based on single-bit-flip fault model, up to 80% of transient faults in loop induction variables would lead to process crashes. A multi-bit-flip event would be even more likely to result in a process crash. This result motivated **CARE** to exploit the side effects of code optimization techniques, that are widely deployed in modern compilers to enable the recovery on corruptions in induction variables and therefore improve its overall recovery rate against soft failures.

4.4 Code Optimization Techniques

CARE leverages semi-redundancies introduced by code optimization techniques, including strength reduction and loop unrolling, to enhance its recovery capability against soft failures. These code optimization techniques are widely deployed in modern production compilers to improve the code execution efficiency. In this section, we briefly introduce these techniques with a focus on how they produce semi-redundancies that can be exploited for resilience purposes with a simple example.

4.4.1 Strength Reduction

Strength reduction is a code transformation technique in modern compilers that replaces certain costly instructions with less expensive ones without changing programs' correct-

[2] Only loops are considered is because they consume the majority of computation resources of an application

ness. The classic example of strength reduction is to convert expensive multiplications into efficient additions. Although strength reduction is a global optimization, it is typically applied to computations in loops, since most of a program's execution time is typically spent in a small section of code which is often inside loops that is executed over and over. (Similarly, this portion of code is also more highly likely to experience transient errors.) Strength reduction looks for expressions involving a loop invariant value (a value that doesn't change within the body of the loop) and an induction variable (a value which is changing by a known amount in each iteration of the loop). If applicable, strength reduction will transform these expressions into an equivalent but more efficient form. For illustration consider Figure 4.4b which shows the transformed code after applying strength reduction on the original code in Figure 4.4a. As shown in the figure, the original multiplication operation `c * i` is replaced with (reduced to) a cheaper addition operation `k + c`, so the performance of the code is improved. However what's important for **CARE** is that the introduced new expression `k + c` shares a similar computation pattern to `i++`. This provides an opportunity to recover the value of `i`, if it is corrupted, by referring to `k` as long as the initial and step values of these two variables and their updates are available. In particular, the correct value for `i` can be recomputed as `i = k / c` if `k` is in-tainted (The initial values for `i` and `k` are 0, and their step sizes are 1 and c respectively).

4.4.2 Loop Unrolling

In addition to strength reduction, loop unrolling is another compiler optimization technique that could introduce semi-redundancies to codes. The main goal of loop unrolling is to increase a program's speed by reducing (or eliminating) instructions that control the loop (such as end-of-loop tests on each iteration), reducing branch penalties, and hiding latency (e.g., the delay in reading data from memory). To eliminate these computational overheads, loop unrolling re-writes the loop as a repeated sequence of similar independent statements. Figure 4.4c shows the transformed code after applying loop unrolling on the original code

```
1  c = 7;
2  for (i = 0; i < N; i++) {
3      y[i] = c * i;
4  }
```

(a) Original Example Code

```
1  c = 7, k = 0;
2  for (i = 0; i < N; i++) {
3      y[i] = k;
4      k = k + c;
5  }
```

(b) Transformed code using Strength Reduction

```
1  c = 7;
2  for (i = 0; i < N; i+=2) { // assume N%2 = 0
3      y[i] = c * i;
4      y[i+1] = c * (i + 1);
5  }
```

(c) Transformed code using Loop Unrolling (For simplicity of illustration, here we assume N % = 2. More codes were generated to handle corner case in production compilers)

Figure 4.4: Semi-redundancy introduced by code optimizations

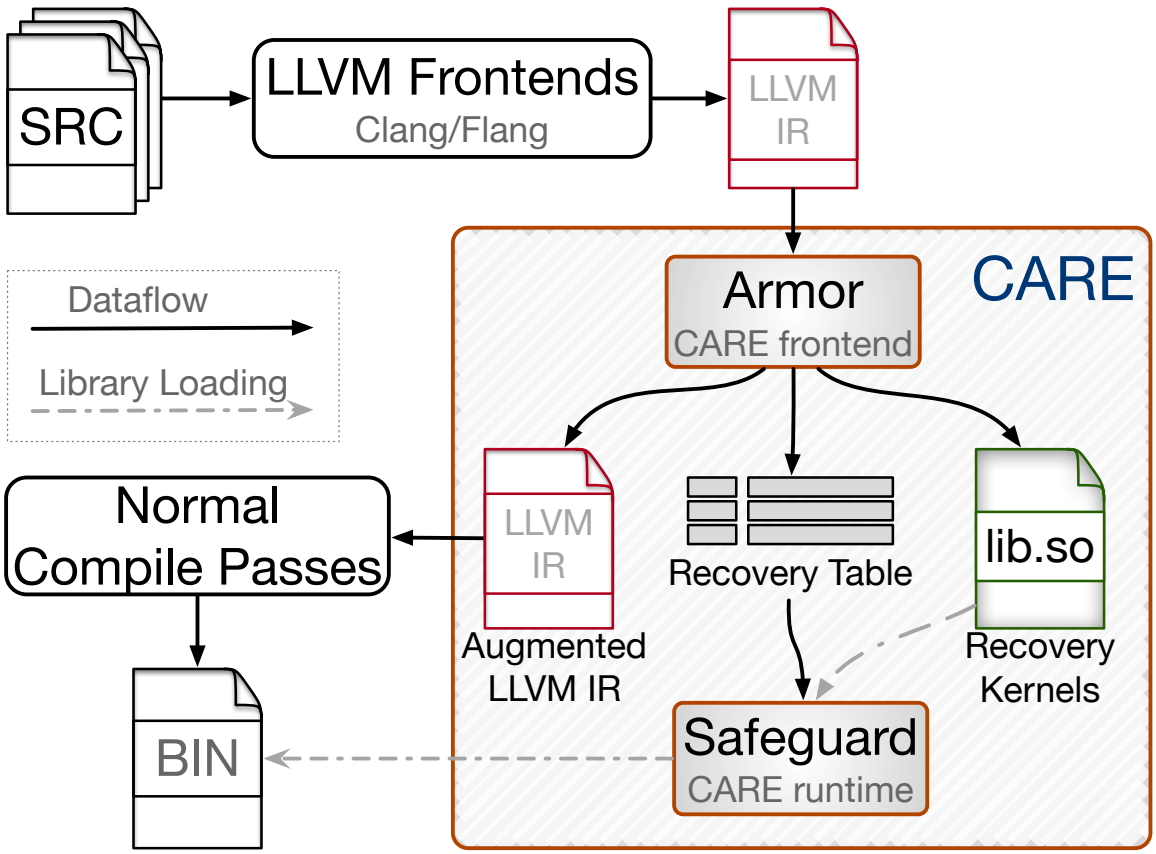


Figure 4.5: Overall architecture of the proposed framework

in Figure 4.4a by unfolding the loop body twice. the transformation reduces the number of end-of-loop tests by almost half in the new code. Meanwhile, it also introduces two similar computing operations, including `i + 2` and `i + 1`. Although unlike the semi-redundancies introduced by the strength reduction in that both of these two instructions depend on variable `i`, they can also be exploited for resilience purposes through additional program analysis and code transformations. While the example code assumes $N\%2 = 0$, this technique can be implemented dynamically even if N is unknown at compile time.

4.5 Overview of the CARE Framework

Given the observations above, we designed the **CARE** environment to focus on recovery from *SIGSEGV* faults. It is a compiler-assisted soft failure recovery framework for scientific applications, and aims to help such applications survive soft failures with negli-

gible overheads. In this section, we depict the overall architecture of **CARE**. As presented in Figure 4.5, the framework consists of two components: 1). **Armor**, a front-end for constructing *recovery kernels*, and 2). **Safeguard**, a runtime system for diagnosing the failure and repairing the corrupted architecture state when a soft failure happens.

Armor is a LLVM pass, and works on LLVM IR representation. It constructs recovery kernels for memory access instructions during the compilation of applications. For each memory access instruction, **CARE** will construct a *recovery kernel* by strategically extracting instructions involved in its address computation. To minimize the overhead of **CARE**, **Armor** compiles recovery kernels into a stand-alone shared library, which will be dynamically loaded by **Safeguard** as needed to repair invalid memory access errors. At the same time, **Armor** also generates a **Recovery Table** which can be considered as the meta-data of recovery kernels. It contains information about how to access and execute a recovery kernel, which is needed by the runtime system. **Safeguard** itself is designed and implemented as a shared library as well. It will be automatically loaded when a process is launched by setting the *LD_PRELOAD* environment variable. Upon loading, **Safeguard** will overload the default *SIGSEGV* signal handler of processes to provide recovery service. Besides such initialization work, **Safeguard** is not activated unless a soft failure occurs. The small load-time overhead of installing a signal handler and the tiny memory overhead for storing the signal handler are its only impact on an application's execution if a fault does not occur. Therefore, **Safeguard** will incur almost negligible overheads during the normal execution of applications. Upon a invalid memory access error, **Safeguard** will be activated by the operating system to diagnose which instruction caused the invalid memory access. It will disassemble the failed instruction to determine which operand is referring to a memory address. And based on the address of the instruction, it will then search, load and execute the related recovery kernel via the help of **Recovery Table** to recompute the accessed memory address for the instruction, and update the related operand. To successfully execute a recovery kernel, **Safeguard** will fetch the values of its parameters from the

```
1  %idxprom156 = sext i32 %i144.0 to i64
2  %arrayidx157 = getelementptr i32, i32* %7, i64 ↩
     %idxprom156
3  %44 = load i32, i32* %arrayidx157, align 4
4  %arrayidx159 = getelementptr i32, i32* %8, i64 ↩
     %idxprom156
5  %45 = load i32, i32* %arrayidx159, align 4
6  %add160 = add nsw i32 %44, %45
7  %sub161 = sub nsw i32 %add160, %29
8  %mul162 = mul nsw i32 %add66, %sub161
9  %add163 = add nsw i32 %mul162, %k.0
10 %idxprom164 = sext i32 %add163 to i64
11 %arrayidx165 = getelementptr double, double* %12, i64 ↩
     %idxprom164
12 %46 = load double, double* %arrayidx165, align 8
13 %sub169 = sub nsw i32 %44, %29
14 %mul170 = mul nsw i32 %add66, %sub169
15 %add171 = add nsw i32 %mul170, %k.0
16 %idxprom172 = sext i32 %add171 to i64
17 %arrayidx173 = getelementptr double, double* %12, i64 ↩
     %idxprom172
18 store double %46, double* %arrayidx173, align 8
```

Figure 4.6: LLVM IR Code for the example code in Figure 4.2.

process's address space. In short, **CARE** relies on the availability of such values which can be typically found in persistent locations such as constant pointers or memory or register values of instructions. Although the overall idea of **CARE** is straight-forward, it comes with several challenges. In the rest of the chapter, we will present the design details for each component, as well as challenges we met and addressed in detail.

4.6 Armor: Building Recovery Kernels

Armor is a compiler pass based on the LLVM framework [49]. It works on LLVM IR, a light-weight low-level intermediate representation of programs. There are several existing tools, such as Clang [50], Flang [51], and DragonEgg [52], which can be used to compile applications into LLVM IR codes. Therefore, the new framework is relatively independent

of programming languages, and can support a majority of scientific applications written in C, C++ or FORTRAN. LLVM IR is in static single assignment (SSA) form. Its syntax is similar to MIPS assembly language, except that LLVM IR has unlimited virtual registers. Figure 4.6 shows an example LLVM IR code for the stencil code in Figure 4.2. Each LLVM IR instruction defines a new value which is used by other instructions. In LLVM IR, memory accesses are issued explicitly through either *LoadInst* or *StoreInst* instructions. For these memory access instructions, **Armor** starts from their address operands and works backwardly to identify instructions involved in their address computations. It then clones and organizes these instructions as a recovery kernel, represented as a normal function in LLVM IR code. **Armor** will construct a recovery kernel for each memory access instruction, except those directly loading from (or storing to) an *AllocaInst* (representing a local variable) or a *GlobalVariable*, since they don't involve any address computations. Recovery kernels for an application are generated into a separate LLVM module, which is then compiled into a stand-alone shared library. There are two challenges need to be addressed by **Armor**:

First, it is not easy to build a recovery kernel that can be successfully executed at runtime due to complex interactions of code optimizations. In general, a recovery kernel can repair transient hardware faults that occur in instructions from which it is cloned, defining the **Coverage Scope** for that particular kernel. Therefore, to make a recovery kernel cover more instructions, **Armor** should clone as many instructions as possible. However, code optimizations in compiler's back-ends for assembly code generation interacts with coverage scope in complex ways. In particular, they could make values that are arguments for recovery kernels unavailable at the memory access instruction, because the related register is reused by compiler for storing other values, without being spilled into the stack. In such case, the kernel is useless. Therefore, **Armor** cannot aggressively copy all computations and, during the compiling time of applications, it has to make sure the built kernel be executed successfully at runtime.

Second, as discussed before, corruptions in induction variables rise another challenge. Recovery from such kinds of corruptions is a challenging problem because induction variables typically employ in-place updates. For every update, the old value of the location (e.g., the register) holding the induction variable is overwritten by a new value. Therefore, once corrupted, it is not obvious how to recover a correct value. An intuitive way to address this problem is leveraging checkpoints of the variable itself, e.g. storing a backup of old value before every update. However, this would involve adding instructions leading to higher register demands inside application's inner loop. This will add undesirable run-time overhead to the most critical portions of these applications.

4.6.1 Building Executable Recovery Kernels

To make sure the recovery kernel for an failed instruction is executable, or saying the parameters of the kernel is always available, **Armor** will stop the process of extraction when it meets predefined **Terminal Values**, where the intuition behind **Terminal values** is that they are guaranteed to be found in registers or memory. A formal definition of **Terminal Values** appears further down below. Informally, it is very critical to find correct **Terminal Values** for recovery kernels, since they are inputs to the kernels. When **Safeguard** is activated to repair a fault, these values must be extracted from the process and then provided to the recovery kernel subroutine in order to recreate the correct address. This requires that those values be accessible and not optimized away. However, **Armor** constructs the recovery kernel at LLVM IR level, and some of the LLVM IR values, like many variables in high-level languages, could be optimized away when they are compiled into machine code, particularly when the optimization flag is enabled. To address this challenge, we leveraged liveness analysis during the construction of recovery kernels. For a memory access instruction, the arguments of its recovery kernel should be live at its position. By definition, a variable is live at a particular point in the program if its value at that point will be used along at least one path that originates at the given program point. In particular, we

leverage the following observation which is true about lowering of IR into machine code: if a value is live at a memory access instruction and its use is non-local (outside the current basic block), it is not optimized away by machine dependent passes (such as instruction selection etc) when the LLVM IR codes is lowered into the machine code. Therefore such a value is eligible as a parameter to the recovery kernel. Based on this insight, **Armor** leverages the algorithm in Figure 4.7 to extract address related computations for a memory access instruction. It will stop the process of extracting instructions when it meets one of the following LLVM IR instructions/values, since they imply start-points of the computation:

1. An *AllocaInst* which represents an variable allocated on the function stack.

2. A *GlobalVariable* which represents a global variable allocated on the data section of process.

3. An *Argument* which represents a function parameter.

4. A *PHINode* that represents a loop induction variable.

5. A *CallInst* calling a complex function. We treat the *CallInst* differently based on the complexity of the callee. **Armor** will stop the process of extraction if the callee updates global variables, arguments passed to it (including memory regions pointed by arguments), or allocates new memory regions. In contrast, if the callee is a simple math operator, e.g., *sqrt*, it will be treated as a normal binary instruction.

6. ***Terminal Value***. A Terminal Value for a memory access instruction I, is a LLVM IR instruction/value which is live at I, with at least one of its operands is dead at I and the dead operand cannot be computed from other live instructions/values. Secondly, as stated earlier, the LLVM value which is live should have a non-local use outside the current basic block (which will ensure that the machine-dependent passes will not eliminate or fold the value making it unavailable). Finally, if every operand of

```
1  bool isExpandable(Value *V, Value *MemAccInst) {
2    if (isa<AllocaInst>(V) || isa<GlobalVariable>(V)
3        || isa<Argument>(V) || isa<PHINode>(V) || ↩
           isComplexCalls(V))
4      return false;
5
6    auto operands = getOperands(V);
7    for (op: operands) {
8      if (!isLiveAt(op, MemAccInst) && !isExpandable(op, ↩
         MemAccInst)) return false;
9    }
10   return true;
11 }
12
13 void getParamsAndStmts(Value *MemAccInst,
14                        vector<Value *> &Params,
15                        vector<Value *> &Stmts) {
16   vector<Value *> Workspace;
17   Value *Addr = getAddrOperand(MemAccInst);
18   Workspace.push_back(Addr);
19   while (!Workspace.empty()) {
20     Value *V = Workspace.back();
21     Workspace.pop_back();
22     if (isExpandable(V)) {
23       Stmts.insert(V);
24       auto operands = getOperands(V);
25       for(op: operands) {
26         if (isa<ConstantData>(Op)) continue;
27         Workspace.insert(Workspace.begin(), op);
28       }
29     } else Params.insert(V);
30   }
31 }
```

Figure 4.7: Pseudo code for extracting address computations

a LLVM instruction/value is live or can be computed from other live values, **Armor** can continue the extraction of computations to extend the coverage scope for the kernel.

For illustration, Figure 4.8 presents a computation-dependency graphs among a set of variables. It shows the address computations for the *LoadInst* in node 1. To build a recovery

kernel, **Armor** will start from node 2 and check liveness of $base$ and $offset$. Since the $offset$ is live at node 1, and $base$ can be computed from a live variable gtc_input in node 9, it will continue the extraction and take the instruction in node 2 as a statement in the recovery kernel, and then evaluate node 3 and node 11. The instructions in node 3 and node 11 will be copied as statements too, since both mul and $call$ are live at node 1, and $densityi$ can be computed from node 9. Node 10 is handled similarly to node 11. And node 4 and node 5 are then evaluated respectively. For node 4, its value mul will be considered as a parameter of the recovery kernel, since its operand sub is not live at node 1, and it cannot be also computed from other live variables. Similarly, for node 5, **Armor** will stop the search until it meet node 6 and node 7. Finally, the recovery kernel for the memory access instruction in node 1 will clone the instructions in node 2, node 3, node 5, node 10, node 11, and node 12 as statements and take the values in node 4 (mul), node 6 ($delz$), node 7 ($mzeta$), and node 9 (gtc_input) as parameters of the kernel.

4.6.2 Recovery from Induction Variables

To recover from corruptions in induction variables, **CARE** exploits side effects introduced by code optimization techniques deployed in modern production compilers. The philosophy behind **CARE** is pretty straightforward. For a given induction variable `i`, updated as `i = i + s`$_i$, **CARE** will leverage scalar-evolution analysis to find another induction variable(s) `k` in the same code region, which is a loop, such that `k` is updated with a computation pattern (`k = k + s`$_k$) similar to `i` and `k` is not used with `i` at the same time to compute a memory address (e.g., `y[i+k]`). **CARE** then pairs them together. And `k` is considered as a partner (or co-related induction variable) to `i`, such that if `i` is corrupted by a fault, **CARE** will be able to recover it by referring to `k` (vice versa) based on the following equation:

$$i = \frac{k - k_0}{s_k} \times s_i + i_0 \tag{4.1}$$

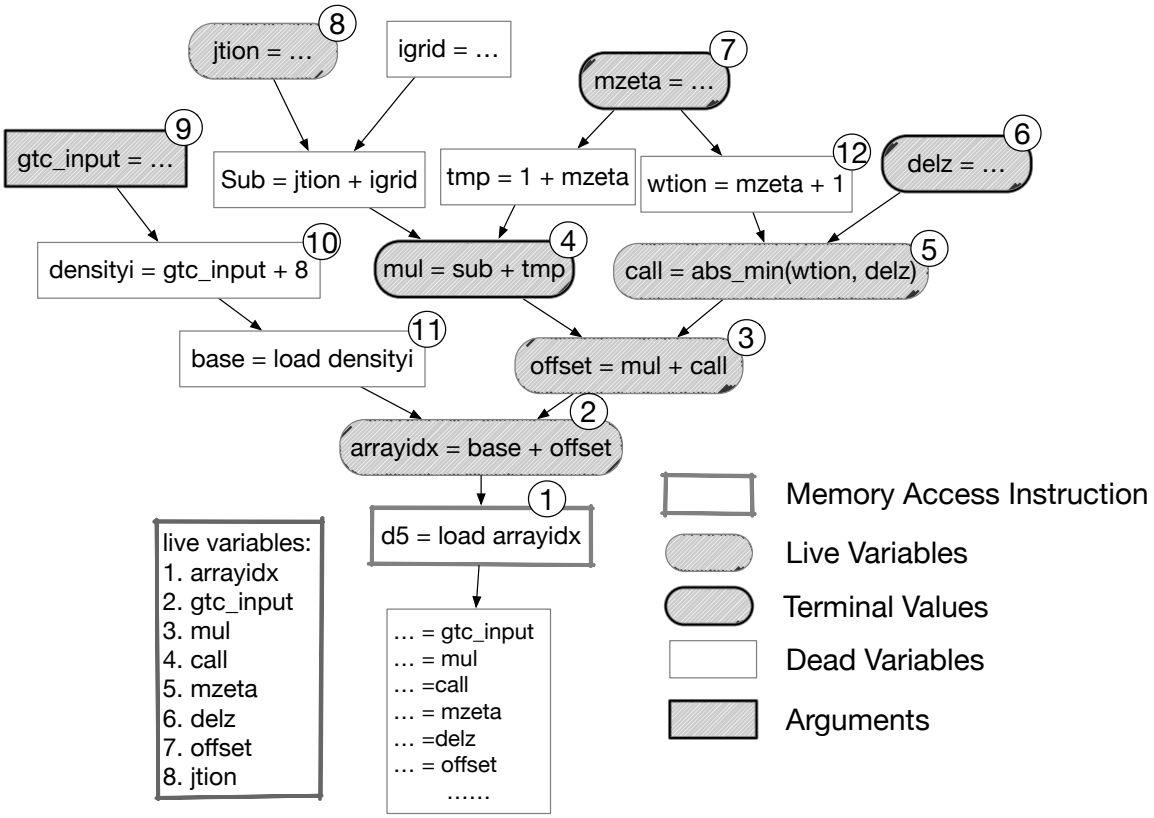


Figure 4.8: The illustration of constructing recovery kernels. $mzeta$ and $delz$ are computed from variables dead at node 1

```
1 for (i = 0; i < N; i++) {
2     sum += *(A++); // from sum += A[i];
3 }
```

Figure 4.9: A sample example.

where, i_0 and k_0 are initial values of i and k respectively. For every induction variable involved in the recovery kernel for a memory access (built in subsection 4.6.1), **CARE** will try to find its pair in the same loop, and add the necessary recovery code to the kernel.

While it would very difficult to find such computation pairs in original source codes, the code optimization techniques deployed in modern compilers, such as strength reduction and loop unrolling, would introduce more opportunities in transformed codes (See section 4.4), which are only accessible by compiler passes. To be able to successfully recover `i` when it is corrupted, **CARE** must know or have accesses to initial values of `i` and `k` and their step sizes at runtime. In other words, when `i` is corrupted, **CARE** should be able to: 1). find its partner `k`; 2) their initial values $\texttt{i}_0$ and $\texttt{k}_0$; 3) their step sizes $\texttt{s}_i$ and $\texttt{s}_k$; and 4) the current value of `k`. If these values are not constant numbers, **CARE** has to make sure they are stored in somewhere (e.g., register or stack) during the code generation pass, such that they are available (the location storing them is not reused by others) regarding less when they are accessed during runtime.

Although semi-redundancies were introduced by the aforementioned code optimization techniques to the transformed codes, they might be not exploitable for resilience purposes due to following challenges:

1. No partner is exposed. In such case, even though these techniques introduced semi-redundancies, but they don't introduce new variables. An example is shown in Figure 4.4c where `i + 1` shares a similar computation pattern to `i += 2`. However, it is useless to **CARE** since they both depend on `i`. In particular, if accessing to `y[i]` failed because of a fault in `i`, there is no partner available for **CARE** to recover it.

2. Sometimes initial values or step sizes are not available at runtime when a failure is

```
1 c = 7;
2 for (i = 0, k=1; i < N; i+=2, k+=2) {
3     y[i] = c * i;
4     y[k] = c * k;
5 }
```

(a) Independent code promotion

```
1 S = A;
2 B = S;
3 for (i = 0; i < N; i++) {
4     sum += *(B++); // from sum += A[i];
5 }
```

(b) Micro-checkpoint

Figure 4.10: Code Transformations in **Armor**. C/C++ are used for illustration only. **CARE** actually works on LLVM IR code.

detected as illustrated in Figure 4.9. In this case, **CARE** would be able to find the partner for `i`, which is `A`, but it may fail to find its initial value A_0. This is because the code generator typically maps `A` to a register, saying `%rax`, and updates it in-place simply with `add %rax, 8`. Therefore, the initial value for `A` is not preserved in applications' process address spaces.

In order to address these problems, **CARE** introduces two additional code transformations, named independent compute promotion (ICP) and micro-checkpoint. For the first situation, **CARE** leverages ICP to transform dependant computations into independent ones, if possible, by introducing new variables. And for "vanishing initial values" challenge, **CARE** introduces code to store (checkpoint) related initial values in the stack, such that they are always available when they are needed for recovering corrupted induction variables. The following subsections present their details.

1. Independent Compute Promotion

Typically, semi-redundancies introduced by loop-unrolling exhibiting in the code in form of *derived induction values* (e.g., `i + 1` in Figure 4.4c). Per discussion before, such semi-redundancies can't be directly exploited by **Armor**, so we introduce compiler pass, ICP, which transforms these derived induction values into independent computations. It will create new induction variables along with their related update instructions to replace original derived induction values. For illustration, Figure 4.10a shows the transformed code derived the code in Figure 4.4c, in which a new variable `k` is created and original `i + 1` is replaced with `k` and `k + 2`. In particular, note that `k` is completely independent from `i`, therefore they can be inferred to recover each other if either one is corrupted. It is worthwhile to note that while ICP does demand an additional register, it doesn't introduce new computation. Such change is often hidden in superscalar processors. Hence, it has negligible penalties to applications' performance.

Algorithm 1 shows the core steps of independent compute promotion. For each loop in LLVM IR codes, ICP iterates over each binary operator in the loop. For those who are directly used (both directly and indirectly) in address computations, **Armor** will create new induction variables to replace them, if they can be expressed in form of (`i = i + s`) based on scalar-evolution analysis, where `s` is a loop invariant value (it doesn't need to be a constant).

2. Micro-checkpoint

Micro-checkpoint is applied only to induction variables whose initial values are not live across the loop body. If a value is not live across the loop body, the location for holding this value could be reused by other variables at runtime, which means it could be not accessible by the recovery mechanism. For these induction variables, **CARE** will checkpoint their initial values into the stack frame by creating new local variables and inserting a store instruction. The transformed code for the code in Figure 4.9 is shown in Figure 4.10b,

Algorithm 1 The Pseudo Code for Independent Compute Promotion.

```
function doIndependencePromotion(loop)
    for every binary operator BO in loop do
        Expr ← getSCEVExpr(BO)
        isAddRec ← isSCEVAddRecExpr(Expr)
        isInAddr ← isUseInAddrCompute(BO)
        if isAddRec && isUsedInAddr then
            initVal ← getStartValue(Expr)
            stepVal ← getStepValue(Expr)
            IndPhi ← createPHINode(initVal)
            Inc ← createIncOp(IndPHI, stepVal)
            IndPhi → addIncomingValue(Inc)
            BO → replaceUsesWith(IndPhi)
        end if
    end for
end function
```

in which a new local variable `S` is allocated to store the initial value (base address) of `A`. And a new variable `B` is introduced as an alias to `A` to iterate over elements in the array. And `B` will be identified as the partner to `i`. While `B = S` looks redundant, but it is not trivial. It provides **CARE** heuristics about where to find initial values for `B`. Notably, the new code has substantially similar performance as the original code, since the instruction insertions are **outside** the loop body. The pseudo code for micro-checkpoint is shown in Algorithm Algorithm 2. It iterates over each induction variable of a loop. If `init` is not a constant number, and it is not live (based on liveness analysis) at the end of corresponding loop, **CARE** will then create a new local variable on the stack to store its initial value.

Please note that while the C/C++ programming language were used for clarity in above examples, **CARE** actually works on LLVM IR codes, which are an intermediate representations of applications.

4.7 Recovery Table: Communicating between Armor and Safeguard

In addition to the recovery library containing the recovery kernel subroutines, **Armor** also generates a **Recovery Table** to describe recovery kernels for **Safeguard**. It contains

Algorithm 2 The pseudo code for micro-checkpoint

```
function DOCHECKPOINTS(loop)
    for every induction variable IV in loop do
        Latch ← GETLOOPLATCH(loop)
        Init ← GETSTARTVALUE(IV)
        Const ← ISCONSTANT(Init)
        Live ← ISLIVEAT(Init, Latch)
        if !Const && !Live then
            Var ← CREATELOCALVARIABLE
            CREATESTORE(Var, IV)
            Val ← CREATELOAD(Var)
            IV → REPALCEALLUSESWITH(Val)
        end if
    end for
end function
```

information about how to access a recovery kernel and which are the parameters to the kernel. The **Recovery Table** is a key-value table as shown in Table 4.5. For each recovery kernel in the recovery library, **Armor** will register an entry for it in the table. The recovery table contains three pieces of information:

1. **key**, which represents an instruction. Each memory access instruction should be associated with a unique key, which will be used to retrieve the related recovery kernel.

2. **symbol**, which represents a recovery kernel. The symbol could be simply the function name of the recovery kernel. It will be used to load the actual implementation of the recovery kernel from the recovery library.

3. **parameters**, which describes the inputs required by the kernel. They are used to retrieve expected input values from the corrupted process.

There are two challenges to be addressed here. First **Armor** and **Safeguard** must agree on the selection of the **key**, which is closely associated with the memory access instruction. For a memory access instruction in LLVM IR and its corresponding assembly instruction in machine code, both **Armor** and **Safeguard** should be able to generate the same key to

Table 4.5: Recovery table for describing recovery kernels

key	symbol	parameters
key1	care_recovery_k1(int16, int, int)	a, b, c
key2	care_recovery_k2(float, int32)	m, n
key3	care_recovery_k3(int8, int64)	d, e

point to the recovery kernel. **Armor** must generate a key at compile-time and associate it with the recovery kernel; meanwhile, **Safeguard** be able to generate the same key using the fault location, and use it to find the recovery kernel. Similarly, parameters are keys to the required input values of the kernel. **Safeguard** relies on them to retrieve input values for the kernel from the process's address space.

Intuitively, the instruction address is a good candidate for the **key**, since each instruction has unique address. It is stable, and easy to get for **Safeguard**. However, it is not available to **Armor**, since it is not generated until the code generation phase. Relying on it would require the modification of the code generation passes in modern compilers. To avoid this complexity, we leveraged the debug information subsystem of modern compilers, which is used to encode source-level program information for machine code. Although **CARE** leveraged this subsystem, it doesn't have to rely on the debug data generated by compiler. In debug data of a program, each instruction is associated with location description, which contains the source file name, the line number and the column number. **CARE** takes the tuple of $(file, line, column)$ as the key to an instruction, since they are accessible both in LLVM IR and in machine code. Specially, **CARE** doesn't require the real debug data of the program, since it won't map instructions to original source-code statements. **CARE** only requires that the debug data is unique for each memory access instruction. **Armor** can generate a fake debug data for each memory access instruction if the debug flag is not enabled. On the other hand, if debug flag is enabled during the compilation, **Armor** needs to resolve the conflicts for some instructions that end up sharing the same debug data. As an additional complexity, since x86_64 assembly supports CISC-style memory

Table 4.6: An example of Debug Information for a local variable. It is simplified for ease of reading.

DW_TAG_Variable	
DW_AT_location	<loclist with 2 entries follows> [0] 0x422cd4∼0x422d3c: DW_OP_reg11 [1] 0x422d3c∼0x422fe4: DW_OP_breg7+4
DW_AT_name	zion3
DW_AT_decl_file	"/path/to/source/file.c"
DW_AT_decl_line	156

access in computations (e.g., "add (%rax, %rcx, 8), %rdx" reads data from memory and adds it to %rdx), some of memory access instructions in LLVM IR might be merged with the related binary instruction during the code generation. Hence, **Armor** also attaches the debug information for memory access instructions to the instructions that directly use their results.

The use of debug mechanism also addresses the second challenge about retrieving arguments for a recovery kernel. For each parameter of a kernel, **Armor** will create a variable description for it by simply assigning a unique name for each parameter. Based on the variable description, the debug information subsystem of the compiler will automatically generate a debug information entry (DIE) to describe the variable in machine code, as shown in Table 4.6. A DIE contains several attributes which are associated with the variable. An important attribute is the "DW_AT_location", which describes the location for a variable. It contains 2 pieces of information, including address ranges and corresponding location of the variable. For example, item $[0]$ in Table 4.6 describes that the variable $zion3$ is located in a register if PC address resides in $[0x422cd4, 0x422d3c)$, and the item $[1]$ describes that the variable is located on the stack at the offset 4 to the frame point register if PC address is in $[0x422d3c, 0x422fe4)$.

4.8 Safeguard: Providing Recovery Service

Safeguard is the runtime system of **CARE**, providing recovery service for applications by setting up and executing recovery kernels constructed by **Armor**. **Safeguard** is built as a shared library, and it is designed to be loaded automatically by setting the $LD\_PRELOAD$ environment variable. Leveraging the "constructor" attribute in modern compilers, **Safeguard** will add a signal handler for *SIGSEGV* immediately after it is loaded. Except this initialization work, **Safeguard** is not activated until a *SIGSEGV* fault occurs. Therefore, it has almost **zero** runtime overhead during the normal execution of applications. To minimize the memory overhead, **Safeguard** only loads recovery kernels when a crash-causing error is detected, and will immediately release the related memory after the repair. Upon a failure, the steps taken by **Safeguard** are shown in Algorithm Algorithm 3. It first retrieves the address for the instruction that issued the *SIGSEGV* signal. Based on the address, **Safeguard** will read the line table of the debug data to get the key for the instruction, and then use the key to find the appropriate recovery kernel from the **Recovery Table**. If successful, it will load the recovery library, retrieve the kernel implementation, decode the debug data to find and retrieve values for parameters, and then execute the recovery kernel. Finally, it will disassemble the instruction, to find its address operand, and update that operand with the value computed by the kernel. If the address operand involves both a base register and a index register, e.g. "mov 8(%rbx, %r8, 4), %eax", **Safeguard** will update the index register (%r8) by default, assuming that index register is computed more frequently than base register, and are more likely to experience faults. It will recompute the value for the index register based on the value in base register, and the value returned by recovery kernel. Before making the actual update, **Safeguard** will check whether the kernel-computed address is the same with the invalid address accessed by the instruction. The update is performed only if they are different. Otherwise, it implies that the fault happened to an instruction that is out of the coverage scope of the recovery kernel, and its argument values

Algorithm 3 The pseudo code for repairing a corrupted process

```
function SAFEGUARD(signo, pcontext)
    Addr ← GETADDROFINSTRUCTION(pcontext)
    key ← GETSRCINFO(Addr)
    kname, params ← SEARCHRECOVERYTABLE(key)
    if no kernel found then
        EXIT(signo)
    end if
    lib ← DLOPEN(libRecovery)
    kfunc ← DLSYM(lib, kname)
    pvalues ← GETPARAMSVALUES(params)
    value ← KFUNC(pvalues)
    operand ← GETADDROPERAND(Addr)
    UPDATE(operand, value)
end function
```

are contaminated. **CARE** lacks of ability to recover such failures. [3]

4.9 Prototype

We implemented a prototype of **CARE** on X86_64 platform and Linux OS. We implemented **Armor** based on LLVM 6.0.1. **Armor** treated some LLVM *CallInst* instructions as a normal binary operators, if they simply call some mathematical kernels, e.g, *sqrt*, or some user-implemented functions that don't update global variables and arguments. But it doesn't clone the implementation of these callee functions, hence, when the recovery kernels are compiled into a shared library, it is necessary to build them with binary source files containing the user-implemented simple functions, and link them with necessary libraries.

On the other hand, **Safeguard** mainly implements a signal handler for *SIGSEGV*. It contains a constructor, which will be executed automatically (by setting *LD_PRELOAD*) to setup/overload the signal handler for processes. The benefit of this design is that **Safeguard** doesn't require source code changes to applications. **Safeguard** computes the address for the failed instruction based on the location where the failure occurs. Failure can happen

[3] Note that a real segmentation fault resulting from program bug or erroneous input will fall into this category as well. **CARE** will declare it non-recoverable and simply propagate the SIGSEGV.

to either an instruction belonging to the application code or to one belonging to the library code. For a failure occurring on applications' code instruction, it will directly use the PC (Program Counter) value corresponding to the failed instruction as the address of the instruction, and for a failure that happened in a shared library, the $offset$, calculated as $PC - base$, is used as the address of the instruction. Here, $base$ is the address at which the library is loaded. This design is mainly restricted due to the differences in mechanisms in terms of encoding the debug data for executable binaries versus the shared libraries. Getting the correct address is the key to retrieve the correct recovery kernel, and related parameters. **Safeguard** utilizes *dladdr* to diagnose the location of failures. The failure location also guides the **Safeguard** to access the correct file for the required debug data.

In addition, **Safeguard** relies on the libdwarf [53] library to read the debug data and the libffi [54] library to execute calls to the recovery kernel. Since "ffi_call" takes pointers as arguments, the address of a variable, instead of a value, is retrieved from the process space. Finally, **recovery table** is implemented based on google protobuf-3.6.0 [55], and the MD5 hash of the debug information tuple $(file, line, column)$ is computed with the mhash [56] library and used as the key.

4.10 Evaluation

This section presents evaluation results of **CARE**. We are mainly interested in the following questions: 1) What is **CARE**'s performance in terms of failure recovery rates?; and 2) What is its overheads to normal executions of applications? In the rest of the section, we will introduce the evaluation methodology and environment, present evaluation results, and discuss the advantages and limitations of **CARE**.

4.10.1 Methodology

We evaluated **CARE** on an X86_64 platform with up to 64 compute nodes, with each node equipped with 48 cores (3072 total cores) and 128GB of memory. We performed fault

injections to emulate transient faults with a methodology similar to that introduced in section 4.2, except that we updated the method of randomly selecting a dynamic instruction, such that injections are only performed to instructions from the application itself and not to library code. In the updated tool, we first profiled the number of executions for each static instruction(from applications only) using the Intel Pin tool. Then we randomly select a static instruction for injection based on the numerical distribution of their executions. Finally, we generate a random number based on the executions of the selected instruction to determine the point at which the fault would be injected. In other words, a dynamic instruction is approximately represented by a pair (I, n), which means the fault will be injected to the instruction I after it is executed n times. In all, we examined around $1000 \sim 2000$ injections that led to *SIGSEGV* errors. In the discussions below it is important to note that **CARE** is largely insensitive to the exact fault model in use. Different choices for fault models would likely change the relative ratios of fault outcomes (such as provided in section 4.2), but if a fault triggers a soft failure, the number of corrupted bits will not impact **CARE**'s operation. Only the actual location of the fault will impact whether or not **CARE** can recover from it.

4.10.2 Workloads

We mainly evaluated **CARE** with 4 scientific workloads including GTC-P, HPCCG, miniMD and CoMD, as well as the NPB benchmark suite. Table 4.7 presents a brief introduction for these workloads. We skipped miniFE since it heavily relies on the C++ STL library which is not fully supported in current prototype. For each workload, the compile-time overheads spent on building recovery kernels and the statistical information about the recovery libraries are presented in Table 4.8. As shown in the table, **Armor** only takes a few seconds to analyze the program and construct the recovery kernels. More than 90% of the overheads were spent on liveness analysis. Despite this offline compile-time overheads shown, **CARE** incurs almost **zero** runtime overhead during the normal execution of the

Table 4.7: Scientific workloads from different scientific domains and implementing different algorithms

Workload	Language	Description
HPCCG	C++	A simple conjugate gradient benchmark code for a 3D chimney domain on an arbitrary number of processors.
CoMD	C	A reference implementation of typical classical molecular dynamics algorithms and workloads as used in materials science.
miniMD	C++	A simple, parallel molecular dynamics (MD) code. It performs parallel molecular dynamics simulation of a Lennard-Jones or a EAM system
miniFE	C++	a Finite Element mini-application which implements a couple of kernels representative of implicit finite-element applications. It assembles a sparse linear-system from the steady-state conduction equation on a brick-shaped problem domain of linear 8-node hex elements. It then solves the linear-system using a simple un-preconditioned conjugate-gradient algorithm
GTC-P	C	A 2D domain decomposition version of the GTC global gyrokinetic PIC code for studying micro-turbulent core transport. It solves the global, nonlinear gyrokinetic equation using the particle-in-cell method.
NPB	C	The NAS Parallel Benchmark (NPB) suite is a small set of programs derived from computational fluid dynamics (CFD) applications. It consists of 5 kernels and 3 pseudo-applications. In this work, NPB3.0-C version is used.

workloads without faults, since it only calls "sigaction" to register signal handlers, which takes a few microseconds. And the memory overheads of **CARE** is fixed to 27MB ($< 1\%$ for evaluated workloads), which is mainly occupied by partial of LLVM and protobuf libraries used by **Safeguard** for encoding/decoding recovery tables. We believe this overhead is negligible for scientific applications that are typically with gigabytes of memory footprints. In this section, we mainly focus on evaluating the fault coverage and recovery time of **CARE**. For these workloads, faults are injected to instructions from applications [4].

[4] **CARE** relies on source code to build recovery kernels, and recovery from transient faults that occur in library code requires the recompilation of libraries from their source codes leveraging **CARE**, which is beyond the scope of our current work, IR recovery binary is the key.

Table 4.8: Statistics of Recovery Kernels

	# of Kernels	Avg. # of IR Insts	Normal Compile Time (s)	**Armor** Overhead (s)
HPCCG	255	2.51	3.575	1.43
CoMD	1143	2.63	1.486	1.17
miniMD	2611	8.2	4.215	1.52
GTC-P	2786	19.18	2.322	2.67
BT	2474	1.17	1.843	0.11
CG	120	1.84	0.617	0.11
EP	30	2.43	0.53	0.05
FT	171	1.81	0.719	0.14
IS	37	1.62	0.47	0.06
LU	1210	1.5	1.25	1.24
MG	547	4.8	0.84	0.43
SP	1462	1.67	1.36	1.64

4.10.3 Fault Coverage of the Basic Framework

CARE is a process recovery technique, which aims to recover processes from invalid memory access errors caused by transient faults. In this subsection, we evaluated the performance of **CARE**'s basic framework based on fault-injection experiments on single process. The recovery capability for induction variables is disabled in this experiment, since we aim to understand the efficiency of the fundamental framework here. We used GTC-P, HPCCG, miniMD and CoMD. We evaluated **CARE** when these applications are compiled with "-O0" (No-opt) or "-O1" (Opt) flags. To support "-O2" and "-O3", which will perform code vectorizations, our current prototype still needs extra engineering work to encode vector-type parameters in recovery tables.

Figure 4.11 presents the fault coverage of **CARE** on the evaluated workloads. On average, **CARE** can recover 83.54% of injected *SIGSEGV* faults, with up to 96% for HPCCG when it is compiled without optimization. **CARE** achieved such high fault coverage mainly due to the fact that majority of *SIGSEGV* faults manifest quickly, typically within only a few dynamic instructions after they occur. The raw data used in address computations is less likely to get updated during such a short time window, especially in the evaluated workloads where they are infrequently updated at the algorithm level. Therefore, **CARE**

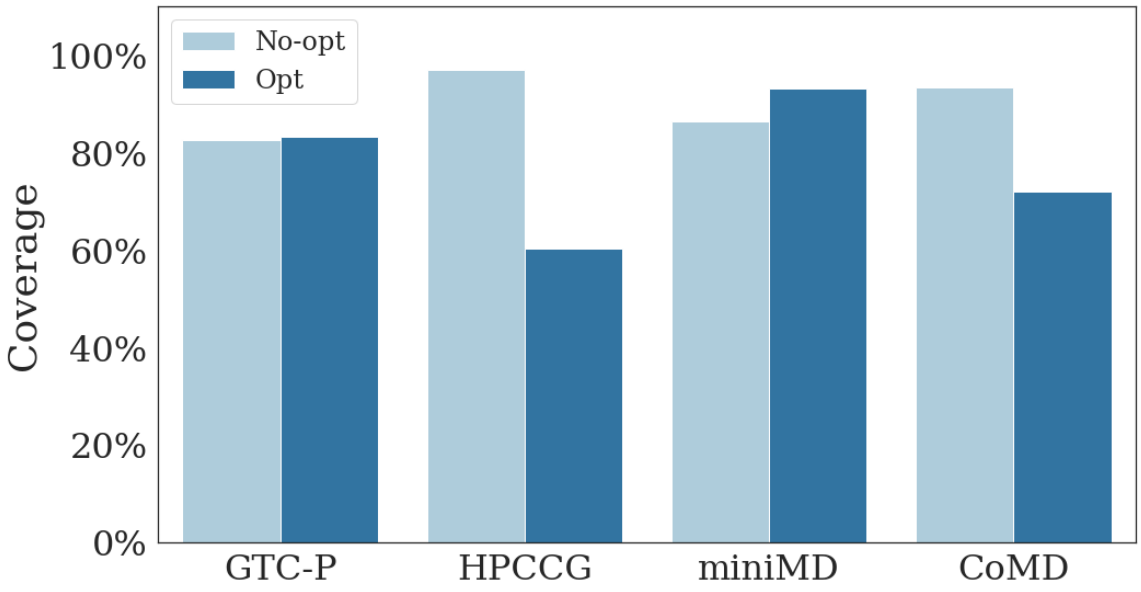


Figure 4.11: Fault Coverage of the Basic Framework of **CARE**.

has a good chance to recompute the addresses. Despite some variance, code optimization didn't introduce significant reduction for the fault coverage of **CARE**. For miniMD, it improved fault coverage by around 7%. this mainly due to code optimization extending the coverage scope of recovery kernels in the miniMD core. This scenario can be illustrated as follows. Figure 4.12 shows two memory accesses. When the code is compiled without optimization, **CARE** can recover errors occurring during the computation of $a + b + c + d$ for case 1, but can only recovery errors occurring in $a + c$ for case 2 because of the immediate update of a and c in line 6 and 7. Code optimization helps to optimize the case 2 to be like case 1 by optimizing out unnecessary memory updates. As a result, the recovery scope of the kernel is extended. Similarly, there is a slight improvement for GTC-P as well. For HPCCG and CoMD, however, code optimization reduced the coverage by 35% and 21%, respectively. For HPCCG, which is a relative simple kernel, a significant portion of dynamic instructions are involved in updating the loop induction variables after the code optimization, therefore they are more likely to be selected by our tool to inject faults. More importantly, because of the code optimization, loop induction variables will be allocated in registers, and updated in-place. If they are corrupted, **CARE** cannot acquire correct values

```
1 int a, b, c, d, *array;
2 array[a+b+c+d];  // case 1
3
4 a += b;
5 c += d;
6 array[a+c];     // case 2
```

Figure 4.12: Code optimization could help extend the coverage scope for case 2 to the same on for case 1

for related recovery kernels, leaving many faults unrecoverable. For CoMD, however, it is mainly because recovery kernels don't have enough coverage scope due to liveness issues.

It is worth noting that during a recovery of failure, **CARE** will not substitute silent data corruptions (SDCs) for failures as is possible with more heuristic based recovery methods [57]. This is because the computation of a recovery kernel is based on the raw data fetched from the process.If raw data is contaminated by a fault, the recovery kernel will definitely generate a wrong address which is the same as the one accessed by the corrupted instruction. Otherwise, **CARE** is guaranteed to get correct address, since it exactly clones the computation from applications.

4.10.4 Contribution of Induction Variable Recovery Scheme

In this subsection, we demonstrate the advantage of **CARE**'s induction variable recovery scheme. Each recovery kernel built by the fundamental framework is enhanced with the recovery codes for involved induction variables. In the rest of this section, we refer this scheme as **IterPro** and continue to use **CARE** as the reference to the fundamental framework. In this experiment, the HPCCG and the NPB benchmark suite were used, since they could be more sensitive to the induction variable recovery scheme based on results in subsection 4.10.3. Similar to subsection 4.10.3, these codes were compiled into LLVM IR codes with clang using the "-O1" flag. An difference is that strength reduction and loop unrolling code optimization passes were also applied. These optimized LLVM IR codes were referred as "baseline" in the rest of the section. **IterPro** works on these "baseline"

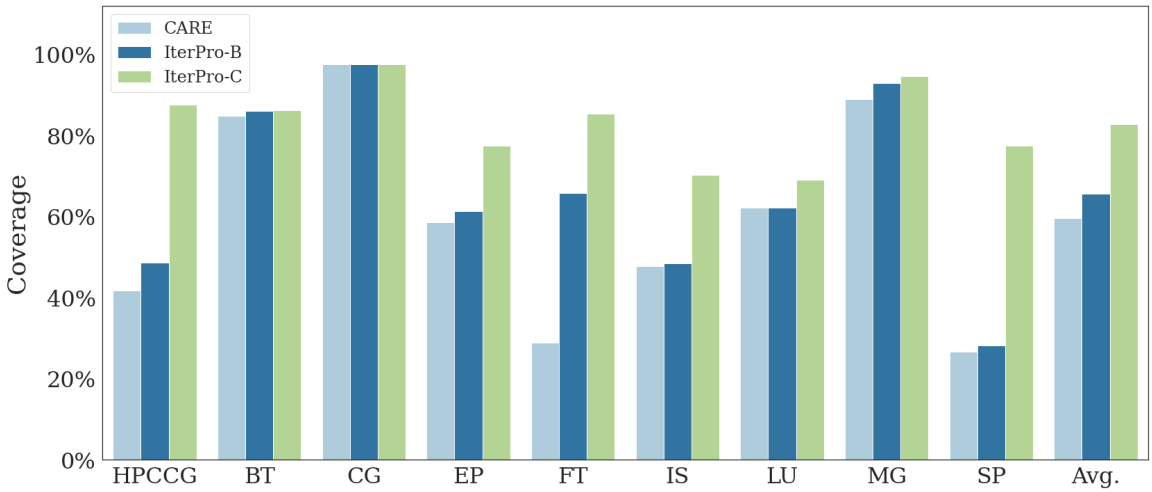


Figure 4.13: Failure Recovery Rates of **CARE** and **IterPro**. It shows the advantage of exploiting side-effects of code optimizations and the efficiency of **IterPro** code transformations.

codes, performing the required code transformations, building recovery kernels and generating the final binaries.

We compared the **IterPro** against the fundamental framework **CARE**. For **IterPro**, we consider two setups: 1) a fractional evaluation when **CARE**'s enhanced code transformations are **not** applied (that is the induction recovery scheme is applied to unaugmented LLVM-generated code), and 2) a comprehensive evaluation when the code transformations are applied. They are respectively labeled as **IterPro**-B and **IterPro**-C. This allows us to both understand the relative contribution of the induction variable recovery scheme over the simpler and fundamental recovery approach of **CARE**, and to see the relative importance of the additional transformation passes in **IterPro** (and the attendant minor instruction additions and register reservations implied by those additional passes).

Figure 4.13 presents the failure recovery rate for each considered scheme [5]. As shown in the figure, for the evaluated workloads, **IterPro**-C improved recovery rate for 8 out of 9 evaluated benchmarks as compared to **CARE**. On average, **IterPro**-C can recover 82.86%

[5]It is necessary to note that, in this experiment, the recovery rate for HPCCG under the **CARE** scheme is different with the recovery rate in subsection 4.10.3 mainly due to the interference from loop unrolling and strength reduction.

of injected *SIGSEGV* faults, while **CARE** recovers 59.7% of these failures. For 3 of them, including FT, SP and HPCCG, it improved the recovery rate by more than $2\times$. On an average, it improved recovery rate by $1.6\times$ across all benchmarks. **IterPro**-C can achieve such significant improvements mainly because of its ability to recover from corruptions in induction variables, which is not enabled in **CARE**. The figure also shows the contribution of **IterPro**'s code augmentations for resilience by comparing the recovery rate of **IterPro**-B and **IterPro**-C. As shown in the figure, the average recovery rate for **IterPro**-B is 65.69%, which is about 6% higher than **CARE**, but around 15% lower than **IterPro**-C. As compared to **IterPro**-B, **IterPro**-C achieved significant improvements ($1.37\times$ on average) for many benchmarks. This is a significant improvement in recovery rates, but unlike **IterPro**-B, those passes do involve code generation changes, hence we must consider their impact on application performance.

The improvement of **IterPro**-C would undoubtedly attribute to the introduced code transformations. These extensions to the normal LLVM code generation are key to the success of **IterPro**-C in that they significantly add to the set of faults from which **IterPro** can recover by introducing "partner" induction variables for some cases where none naturally exist, and storing away necessary initial values where they would not have been otherwise available. In other words, they introduce more "recoverable" introduction variables into codes increasing their resilience. Table 4.9 shows the impact of introduced code transformations by comparing the number of recoverable induction variables in baseline codes and in **IterPro** generated codes. As shown in the table, **IterPro**'s additional LLVM passes increased the number of recoverable induction variables by $4\% \sim 500\%$ (72.65% on average) for 7 out of 9 benchmarks. For two others (EP and IS from NPB), **IterPro**'s additional code transformations introduce a recovery opportunity for induction variables where none existed before (marked by BIG).

In the rest of the paper, **IterPro** is referred as to **IterPro**-C.

Table 4.9: Number of recoverable induction variables respectively in baseline and **IterPro** transformed codes.

Benchmark	# of Loops	Baseline	**IterPro**	Improvement
HPCCG	30	38	43	13.16%
BT	177	253	277	9.49%
CG	38	8	40	500%
EP	12	0	4	BIG
FT	53	46	48	4.35%
IS	7	0	12	BIG
LU	189	340	370	8.82%
MG	81	32	64	200%
SP	316	364	474	30.22%

4.10.5 Recovery Time

Recovery time measures the time required by **Safeguard** to recover from a fault. Clearly a single faulted computation might feed into several memory access instructions. What might not be intuitively obvious is that in this situation, **Safeguard** could be activated several times, recovering the effects of each manifestation of the fault. Figure 4.14 shows that **CARE** can recover a process from a *SIGSEGV* fault with only a few tens of milliseconds. In fact, only a tiny percentage of that recovery time is spent in the generated recovery kernel. They generally only contain a few instructions related to address computations and while their use is key to **CARE**, their actual portion of the recovery time is negligible. In fact, for each activation, more than 98% of the recovery time is spent on preparing the execution of recovery kernels, including diagnosing the failure, loading recovery table and recovery library, and retrieving arguments from stalled process.

4.10.6 Impact on Parallel Jobs

In this subsection, we examine the impact of **CARE** on MPI parallel jobs. Workloads including GTC-P, HPCCG, miniMD and CoMD were used since they have MPI support. We run the workloads with 512 processes and 6 threads on a cluster with 64 nodes (3072 cores). For each run, we injected a **CARE**-recoverable fault to rank 0 of the job. We wrote a wrap-

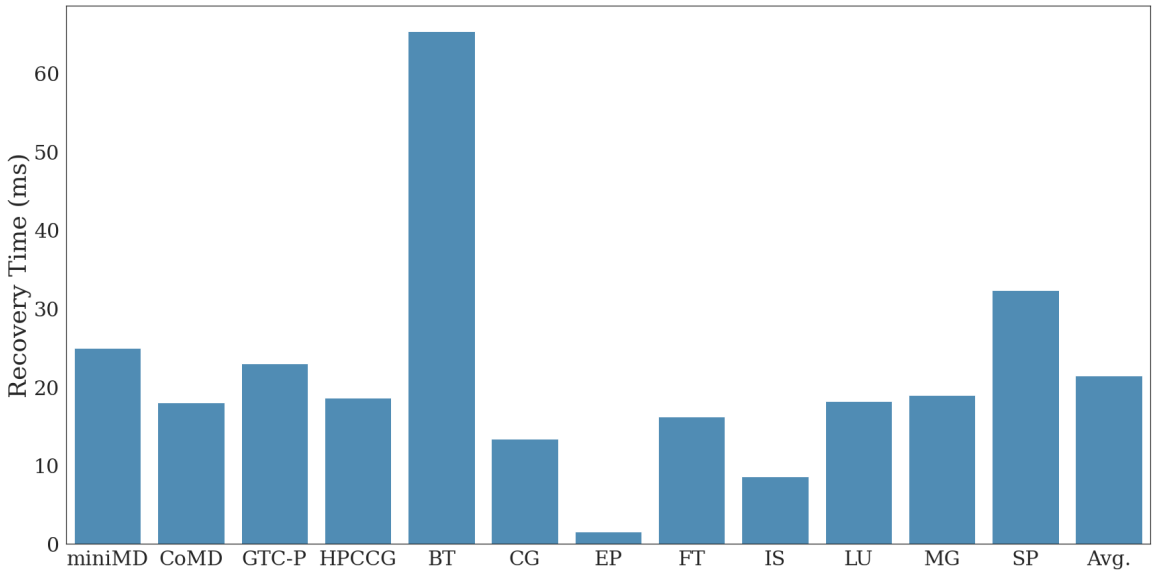


Figure 4.14: Recovery time of **CARE**

per to $PMPI\_Init$, in which an injection process is created and attached to rank 0 using $ptrace$. For a injection point (I, n), the injection process will set a break-point at I, stop the rank 0 after the instruction I is executed n times, and then contaminate the destination operand of the target instruction and continue the execution of process. We performed 100 injections, one injection per-run. Figure 4.15 compares the execution time of parallel jobs when a *SIGSEGV* fault occurred in rank 0 and is repaired by **CARE**. It shows that, despite some execution variance across different runs, **CARE** can almost completely mask the impact of recoverable *SIGSEGV* faults to parallel jobs. It can help parallel jobs to survive invalid memory access errors caused by transient faults. With the protection from **CARE**, parallel jobs when experiencing an invalid memory access error can finish their computations with almost no delays as compared to their normal executions. This is due to the low recovery time of **CARE**. In comparison, even a small run of GTC-P relying upon checkpoint/restart would require at a minimum dozens of seconds (14.367s, 25.946s, or 37.56s on average to recover from a failure if checkpoint is respectively scheduled every 20, 50, or 75 time-steps) to recover from a failure, even if some automatic restart mechanism were available and the new job were to be scheduled immediately.

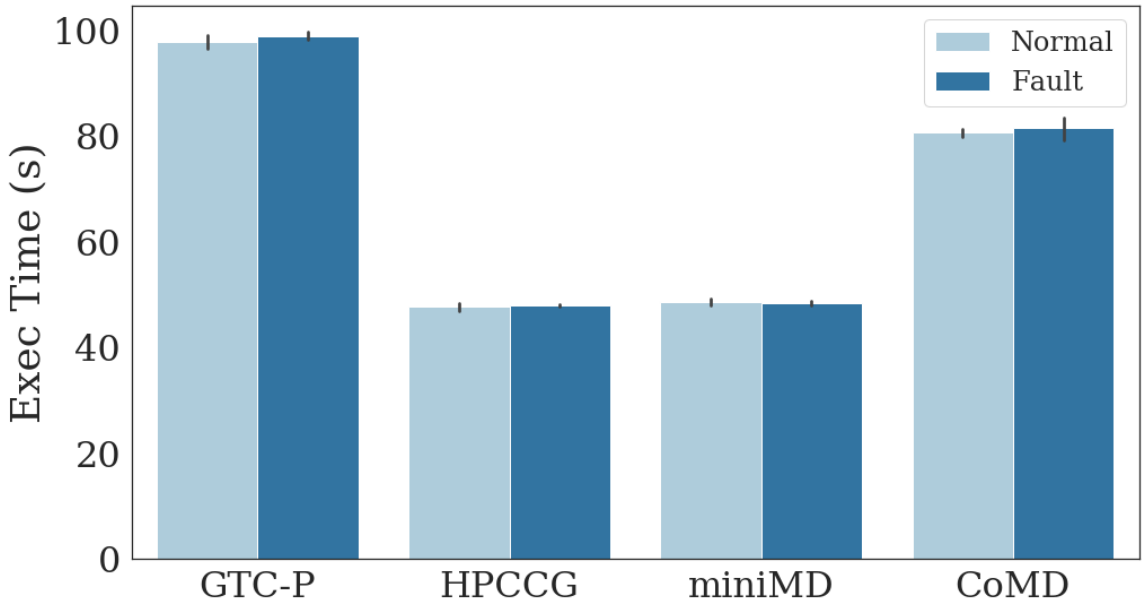


Figure 4.15: Parallel jobs can finish the computations without delays with **SIGSEGV** recovered by **CARE**

4.10.7 Failures in Shared Library – *BLAS*

BLAS is a popular linear algebra library written in FORTRAN. It prescribes a set of low-level routines for performing common linear algebra operations, and is widely used in many scientific applications and machine learning workloads. BLAS routines are categorized into three levels, which correspond to the degree of the polynomial. In this subsection, its preliminary evaluation results are presented.

We rebuilt the *BLAS* from its source code (copied from *LAPACK-3.8.0*) as a shared library ("libblas.so"). The test program sblat1 (in "TESTING/sblat1.f") for REAL level 1 blas, which is provided by the package, was used as the driver to the library. *sblat1* is dynamically linked to "libblas.so". We randomly injected faults into either *BLAS* or *sblat1*. Table 4.10 briefly presents the statistics of recovery kernels for *sblat1* and *BLAS*, and the performance achieved by **CARE** for them. It shows that **CARE** achieved around 83% coverage with almost negligible overheads. We should point out that *sblat1* only covers 12 REAL level-1 routines, which is a small fraction of procedures provided by *BLAS*.

Table 4.10: Statistics and Performance for *sblat1*/*BLAS*

	# Kernels	Normal Compile time (S)	**Armor** Overhead (S)	Coverage	Recovery Time
BLAS	10931	6.89	4.973	83.49%	5.7ms
sblat1	187	1.78	1.98		

While a detailed evaluation for BLAS is currently underway, the preliminary results in this subsection demonstrate that **CARE** effectively supports the recovery of failures which could occur in libraries with good coverage, small overheads and recovery times.

4.11 Related Work

Detection and recovery from failures are not new topics in HPC and other environments [41, 43, 20, 21, 58], so there is significant prior work to consider. In this section, we present a brief survey of studies most related to **CARE**.

Studies in [18, 19, 21] examined the impact of transient faults on scientific applications. Their results showed that a significant portion of transient faults could manifest as soft failures. This motivated us to study how soft failures manifest inside scientific applications and whether there are common features that can be explored to design an efficient resilience mechanism for them, resulting in the design of **CARE**. Georgakoudis et al. [59] and Chang et al. [60] designed and evaluated new fault injection tools for transient faults. While these tools are valuable to the community, they are not a good fit for **CARE** because they either work on high-level intermediate representations (LLVM machine IR) which is inaccurate as compared binary-level injections, or incur high overheads (3$\times$ slowdown) making it infeasible to run large scale ($\sim$100 000 injections) fault injection experiments.

Besides **CARE**, there are several studies on online recovery from process failures such that applications can continue their normal executions. Rx [61] aims to recover from a process failure by rolling applications back to a previous safe status, and then continuing its execution with a minor modification to its environment. Rx is motivated by the observation that many program bugs are associated with the setup of process environments,

so changing the environment setup could avoid the crashes. Its techniques could help handle transient faults by simply replaying the computation *without* changing the environment, however its basic operation requires at least partial application checkpoints which are likely to have significant cost. RCV [57] is another online failure recovery technique for divide-by-zero (*SIGFPE*) and null-dereference (*SIGSEGV*) errors. RCV's approach explores a set of heuristics for recovery. For instance, it returns zero as the default result of the divide for divide-by-zero errors, discards invalid write instructions that accessing near-to-zero addresses and returns 0 for invalid read operations. These techniques are computationally inexpensive and may succeed in getting the application to continue, but are likely to introduce SDCs as a side effect. LetGo [20] shares a similar idea to RCV, and is specially designed for handling soft failures in scientific applications. Its recovery strategy employs a set of heuristics too. Upon a failure, it will reset architecture states to a pre-defined value, and then continue the execution of the application. Obviously such heuristic based method could lead to SDCs, which is another challenge problem in HPC community.

In contrast, **CARE** undertakes a proper recovery process with regards to the maligned address computation by recomputing it as per the program semantics and through the use of in-tainted values by synthesizing a very lightweight function. It develops careful correspondence mechanism to co-relate the recovery handlers to the fault causing instruction at runtime. While **CARE** shares the similar goal and design to RCV and LetGo in that they all aim to help applications to survive failures by replacing the default signal handler with their own one to provide recovery services, **CARE**'s approach is more accurate than others, and will not introduce SDCs.

4.12 Conclusion

In this chapter, we present and evaluate **CARE**, a lightweight and compiler-assisted error-recovery mechanism that allows processes to survive crashes caused by certain transient faults, such that applications can continue their execution. Based on our experimental stud-

ies, we identified *SIGSEGV* as a major symptom to soft failures, and **CARE** is designed for repairing *SIGSEGV* errors. For each memory access instruction that involves complex address computations, **CARE** first builds a recovery kernel by cloning its address computations. At runtime, it maps the fault causing instruction to a failure recovery handler which recomputes the address and masks the fault. **CARE** exploits semi-redundancies introduced by modern compiler optimization techniques, including strength reduction and loop unrolling, for resilience purposes. Coupled with two new code transformations, **CARE** leverages these semi-redundancies to repair soft failures caused by faults in induction variables. We evaluated **CARE** with four scientific workloads and the NPB benchmark suite. During their normal executions, **CARE** incurs almost **zero** runtime overhead and fixed 27MB memory overheads, and it can recover averagely to 83% *SIGSEGV* faults within a few milliseconds.

We also demonstrated the impact of **CARE** on parallel jobs at the scale of 3072 cores. Due to the low-latency recovery mechanism of **CARE**, the parallel jobs can finish their computations as normal with almost no delays, even if a crash-causing error occurred and repaired by **CARE** during their executions. These results show that **CARE** can allow applications to survive some classes of transient failures with little or no overhead, which is of particular benefit in HPC scenarios where the cost of application failure is high. **CARE** could also improve MTBF and therefore could open a door towards research on relaxation of the checkpoint frequency which could have significant resource and performance implications.

CHAPTER 5
CONCLUSION

This chapter first summarizes this dissertation and highlights its contributions on the design and implementation of light-weight resilience techniques against transient hardware faults via the help from compiler techniques. It concludes the dissertation with a discussion on research directions for future work.

5.1 Summary

Resilience is projected to be a critical challenge for HPC systems due to system scaling trends in higher circuit density, smaller transistor size and near-threshold voltage (NTV) operations. These technology trends would make the system more susceptible to transient hardware faults caused by such things as high-energy particle strikes and heat flux. Transient hardware faults would either lead scientific applications to generate incorrect outputs (SDCs), or crash the execution of an application (Soft Failures). SDC is harmful to scientific discoveries, since it could lead to incorrect scientific insights; and the presence of soft failures requires the application to recompute the lost computation before continuing the execution, necessitate more frequent checkpoints than would otherwise be desirable, and lead to significant system overheads. Considering the fact that, despite the increasing threat from transient hardware faults, fault-free operations would remain the most common case. Therefore, desirable solutions against transient hardware faults call for low (no) overhead systems that do not compromise the performance under no-fault conditions. To this end, my dissertation presents a compiler-assisted resilience framework to validate a given simulation run to be free of SDCs and allows processes to survive soft failures instead of being simply terminated and restarted. A summary of techniques proposed in the dissertation are presented in below:

1. It presented and evaluated LADR, a lightweight application-level approach to protect applications from SDCs. LADR employs anomaly-based techniques for detecting SDCs in state variables of scientific applications. It improves extant anomaly-based techniques by focusing on minimizing runtime and memory overheads primarily through exploiting correlation among state variables and data points. LADR is evaluated using application-level fault injection experiments. Results suggest that LADR can protect application from influential SDCs with no more than 8% overheads. For two of evaluated workloads, it only incurs around 2% overheads. LADR demonstrates that it is unnecessary to apply anomaly detection techniques on all crucial variables. Instead, a subset of crucial variables can be identified employing compile-time data-flow analysis.

2. It introduced **CARE**, a lightweight and compiler-assisted error-recovery mechanism that allows processes to survive certain soft failures, such that the applications can continue their execution. To motivate the design of **CARE**, it enthusiastically studied the manifestation of soft failures based on experimental fault injection experiments, and identified *SIGSEGV* as a major symptom to soft failures. Thus, **CARE** is specially designed for repairing *SIGSEGV* errors. For each memory access instruction from scientific applications, **CARE** first builds a recovery kernel by cloning its address computations. At runtime, it maps the fault causing instruction to a failure recovery handler which recomputes the address and masks the fault. We evaluated **CARE** mainly with a set of scientific workloads. During their normal executions, **CARE** incurs almost **zero** runtime overhead and fixed 27MB memory overheads, and it can recover averagely to 83% *SIGSEGV* faults within a few milliseconds. We also demonstrated the impact of **CARE** on parallel jobs at the scale of 3072 cores. Due to the low-latency recovery mechanism of **CARE**, the parallel jobs can finish their computations as normal with almost no delays, even if a crash-causing error occurred and repaired by **CARE** during their executions. These results show that

CARE can allow applications to survive some classes of transient failures with little or no overhead, which is of particular benefit in HPC scenarios where the cost of application failure is high. **CARE** could also improve MTBF and therefore could open a door towards research on relaxation of the checkpoint frequency which could have significant resource and performance implications.

3. It demonstrates a valid way of exploiting code optimization techniques, including strength reduction and loop unrolling, for soft failure recoveries. During the compilation of applications, these code optimization techniques would introduce equivalent computation patterns and values (semi-redundancies) into the code, providing opportunities which can be exploited for resilience purposes. It introduced two new code transformations to explicitly expose these semi-redundancies to underlying recovery kernel builders. It demonstrated that, by taking advantages of these semi-redundancies, the recovery rate for soft failures can be increased by up to $2.9\times$. To the best of our knowledge, it is the first technique that explores side-effects of code optimization techniques for resilience purposes.

5.2 Future Work

The techniques introduced in this dissertation opened the door of exploiting applications' properties for lightweight resilience solutions against transient hardware faults. While the evaluation results demonstrate that they achieved remarkable milestones, there are still many open challenges. We propose 3 potential future research directions:

5.2.1 Quantifying Propagation Impacts and Error Bounds of SDCs

LADR introduced in chapter 3 reveals that the data-flow among state variables of scientific applications implies the propagation path of SDCs, and this information can be used to minimize the runtime overhead of existent data-anomaly-based SDC detection methods.

Despite its efficiency, we believe addressing the following two questions would make this approach more attractive to the community:

1. Quantify Propagation Impacts. The data-flow information exploited by LADR only qualitatively depicts the fault propagation path of SDCs. Assuming a fault corrupted a value in state variable `a`, and this fault will be propagated into state variable `b` when `b` is updated using `a`. Will the impact of the SDC be amplified or diminished during this propagation? An SDC is said to be amplified if the introduced error in `b` is larger than the error in `a` (Here, an error is defined as $(V_{faulty} - V_{normal})/V_{normal}$), otherwise, it is diminished. Intuitively, if `b` is selected as the sink variable to `a`, we expect the SDC impact would be amplified from `a` to `b` to make sure the SDC is detectable. Augmenting the data-flow graph with this type of information will help to guide the selection of sink variables such that SDC is amplified along the propagation paths, therefore improve/guarantee the fault coverage of SDC detection.

2. Quantify User-acceptable Error Bounds. Data-anomaly based detection methods targets on influential SDCs. However, it still lacks of quantitative definition of "influential impacts" in current framework. On the other hand, different applications could have vary sensibility to SDCs. Therefore, it is necessary to accept user-defined acceptable error bounds into current SDC detection framework.

5.2.2 Resilient Code Generation Techniques

In chapter 4, **CARE** was depicted as a soft failure recovery framework based on compiler techniques. Meanwhile, it is also observed that the code optimization passes in modern compiler could interact with the recovery rate of **CARE** in a complex way: some compiler passes, although may need extra code transformations, could bring opportunities for resilience techniques; and some other may not. Although **CARE** has exploited benefits of a limited set of code optimization techniques, including strength reduction and loop unrolling, there still lacks of comprehensive studies of code optimization techniques for

resilience purposes. In addition, researchers in [62] demonstrated that many code shapes in modern scientific application could exhibit nature error resilience capability against transient hardware faults. Based on these two observations, an interesting question is how to generate code to improve its resilience while maintaining its performance, given that almost all existent compiler passes are primary designed for improving code performance, but not the resilience ability.

5.2.3 Recovery based on Idempotent Processing

Almost all existent SDC detection solutions are completely based on software-level approaches. While they have demonstrated their efficiency for SDC detection, due to the design strategy, they also have a extremely long detection latency (except instruction duplication based approaches which will incur significant runtime overheads), which have posed significant challenges for designing corresponding recovery methodologies. For example, the detection latency for anomaly-based SDC detection methods, such as LADR, is measured in several time-steps of simulations (or millions or even billions of instructions), therefore, they all currently have to rely on the expensive checkpoint/restart mechanism for recoveries. Thus, efficient resilience techniques demand low-latent fault detection methods.

On the other hand, Gaurang et al. [63, 64] presented a low-cost hardware detector for transient hardware faults based on acoustic sensors. As compared to current software level approaches, it has very low detection latency. This approach opened a way of exploiting the cooperation between hardware and software for resilience approaches against transient hardware faults. Qingrui et al. [65] exploited this hardware feature, and proposed a micro-checkpoint method, such that processes will periodically save their architecture states and upon a detection of transient fault by the hardware, the impacted process can be safely rolled back to a checkpoint to repair the corrupted process status. As compared to existing checkpoint techniques, micro-checkpoint only incur $5\% \sim 10\%$ performance overheads.

However, the micro-checkpoint may be unnecessary in this scope, since applications

can be safely divided into a set of idempotent regions [66]. An idempotent region is a sequence of code that can be safely re-executed several times without changing the result. Therefore, if a fault corrupted the computation inside a region, it is safe to repair the process state by rolling back to the beginning of region. A big challenge for designing such idempotent region based resilience approaches comes from the requirement of coordinating the region size to the detection latency of the hardware detector. The idempotent regions constructed by current algorithms [66, 67, 68] are too small to benefit from the current hardware detector for transient hardware faults. Each idempotent region constructed by these algorithms typically contains a few instructions operating on scalar variables, which is could be shorter than the detection latency of the hardware detector. Therefore, if a fault corrupted the computation is a region, it may be not detected until the process has executed into other regions, which would be difficult to be repaired. Therefore, compiler techniques for constructing relatively coarse-granularity (larger) idempotent regions would be interesting and helpful for these kinds of approaches.

REFERENCES

[1] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "C3: A system for automating application-level checkpointing of mpi programs," in *Languages and Compilers for Parallel Computing*, L. Rauchwerger, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 357–373, ISBN: 978-3-540-24644-2.

[2] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," *Journal of Physcs: Conference Series*, vol. 46, Jul. 2006.

[3] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10, Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11, ISBN: 978-1-4244-7559-9.

[4] X. Ni, E. Meneses, N. Jain, and L. V. Kalé, "ACR : Automatic Checkpoint / Restart for Soft and Hard Error Protection," in *Proceedings of International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2013.

[5] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011.

[6] S. Amarasinghe, D. Campbell, and W. C. etc., "Exascale software study: Software challenges in extreme scale systems," DARPA IPTO, Air Force Research Labs, Tech. Rep., 2009.

[7] M. H. Wright and Al., *The opportunities and challenges of exascale computing*, 2010.

[8] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 4, pp. 374–388, Nov. 2009.

[9] S. Hukerikar and R. F. Lucas, "Rolex: Resilience-oriented language extensions for extreme-scale systems," *The Journal of Supercomputing*, vol. 72, no. 12, pp. 4662–4695, Dec. 2016.

[10] D. Oliveira, L. Pilla, N. DeBardeleben, S. Blanchard, H. Quinn, I. Koren, P. Navaux, and P. Rech, "Experimental and analytical study of xeon phi reliability," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17, Denver, Colorado: ACM, 2017, 28:1–28:12, ISBN: 978-1-4503-5114-0.

[11] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 29–40.

[12] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello, "Optimization of multi-level checkpoint model for large scale hpc applications," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 1181–1190.

[13] N. Sasaki, K. Sato, T. Endo, and S. Matsuoka, "Exploration of lossy compression for application-level checkpoint/restart," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 914–922.

[14] P. Fernando, S. Kannan, A. Gavrilovska, and K. Schwan, "Phoenix: Memory speed hpc i/o with nvm," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, IEEE, 2016, pp. 121–131.

[15] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," in *Proceedings of International Symposium on Computer Architecture*, 2003.

[16] M. A. Heroux, "Toward resilient algorithms and applications," in *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale*, ser. FTXS '13, New York, New York, USA: ACM, 2013, pp. 1–2, ISBN: 978-1-4503-1983-6.

[17] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory errors in modern systems: The good, the bad, and the ugly," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15, Istanbul, Turkey: ACM, 2015, pp. 297–310, ISBN: 978-1-4503-2835-7.

[18] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, Salt Lake City, Utah: IEEE Computer Society Press, 2012, 57:1–57:11, ISBN: 978-1-4673-0804-5.

[19] C.-Y. Cher, M. S. Gupta, P. Bose, and K. P. Muller, "Understanding soft error resiliency of bluegene/q compute chip through hardware proton irradiation and software fault injection," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, New Orleans, Louisana: IEEE Press, 2014, pp. 587–596, ISBN: 978-1-4799-5500-8.

[20] B. Fang, Q. Guan, N. Debardeleben, K. Pattabiraman, and M. Ripeanu, "Letgo: A lightweight continuous framework for hpc applications under failures," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17, Washington, DC, USA: ACM, 2017, pp. 117–130, ISBN: 978-1-4503-4699-3.

[21] J. Calhoun, M. Snir, L. N. Olson, and W. D. Gropp, "Towards a more complete understanding of sdc propagation," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17, Washington, DC, USA: ACM, 2017, pp. 131–142, ISBN: 978-1-4503-4699-3.

[22] T. J. Slegel, R. M. Averill, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, *et al.*, "Ibm's s/390 g5 microprocessor design," *IEEE micro*, vol. 19, no. 2, pp. 12–23, 1999.

[23] L. Spainhower and T. A. Gregg, "Ibm s/390 parallel enterprise server g5 fault tolerance: A historical perspective," *IBM Journal of Research and Development*, vol. 43, no. 5.6, pp. 863–873, 1999.

[24] R. W. Hamming, "Error detecting and error correcting codes," *The Bell system technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

[25] K. Ferreira, J. Stearley, J. H. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2011.

[26] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining partial redundancy and checkpointing for hpc," in *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ser. ICDCS '12, Washington, DC, USA: IEEE Computer Society, 2012, pp. 615–626, ISBN: 978-0-7695-4685-8.

[27] N. Oh, S. Mitra, and E. J. McCluskey, "Eddi: Error detection by diverse data and duplicated instructions," *IEEE Trans. Comput.*, vol. 51, no. 2, 2002.

[28] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of International Symposium on Code Generation and Optimization*, 2005.

[29] E. Berrocal, L. B. Gomez, D. Sheng, Z. Lan, and F. Cappello, "Lightweight silent data corruption detection based on runtime data analysis for HPC applications," in *Proceedings of International Symposium on High-Performance Parallel and Distributed Computing*, 2015.

[30] S. Di and F. Cappello, "Adaptive Impact-Driven Detection of Silent Data Corruption for HPC Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2809–2823, Oct. 2016.

[31] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer, "HAFT - hardware-assisted fault tolerance.," in *Proceedings of European Conference on Computer Systems*, 2016.

[32] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[33] N. Hübbe, A. Wegener, J. M. Kunkel, Y. Ling, and T. Ludwig, "Evaluating Lossy Compression on Climate Data.," in *Proceedings of International Supercomputing Conference*, 2013.

[34] L. A. B. Gomez and F. Cappello, "Detecting and correcting data corruption in stencil applications through multivariate interpolation," in *Proceedings of International Conference on Cluster Computing*, 2015.

[35] C. Engelmann, H. H. Ong, and S. L. Scott, "The Case for Modular Redundancy in Large-Scale High Performance Computing Systems," in *the IASTED International Conference*, 2009.

[36] Z. Chen, S. W. Son, W. Hendrix, A. Agrawal, W. K. Liao, and A. Choudhary, "Numarck: Machine learning algorithm for resiliency and checkpointing," in *Proceedings of International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2014.

[37] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Chicago, IL, USA: IEEE, May 2016, pp. 730–739.

[38] K. S. Yim, "Characterization of impact of transient faults and detection of data corruption errors in large-scale n-body programs using graphics processing units," in *Proceedings of International Symposium on Parallel and Distributed Processing*, 2014.

[39] Z. Chen, R. Inagaki, A. Nicolau, and A. V. Veidenbaum, "Software fault tolerance for fpus via vectorization," in *Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2015.

[40] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight Silent Data Corruption Error Detector for GPGPU," in *Proceedings of International Symposium on Parallel and Distributed Processing*, 2011.

[41] Z. Chen, "Algorithm-based recovery for iterative methods without checkpointing," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11, San Jose, California, USA: ACM, 2011, pp. 73–84, ISBN: 978-1-4503-0552-5.

[42] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," in *Proceedings of International Symposium on Principles and Practice of Parallel Programming*, 2012.

[43] Z. Chen, "Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods," in *Proceedings of International Symposium on Principles and Practice of Parallel Programming*, 2013.

[44] T. Davies and Z. Chen, "Correcting soft errors online in lu factorization," in *International Symposium on High-Performance Parallel and Distributed Computing*, 2013.

[45] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining partial redundancy and checkpointing for hpc," in *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ser. ICDCS '12, Washington, DC, USA: IEEE Computer Society, 2012, pp. 615–626, ISBN: 978-0-7695-4685-8.

[46] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose, "Understanding the propagation of transient errors in hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, Austin, Texas: ACM, 2015, 72:1–72:12, ISBN: 978-1-4503-3723-6.

[47] N. A. Quynh, *Capstone: Next-Gen Disassembly Framework*, 2014.

[48] V. C. Sharma, G. Gopalakrishnan, and S. Krishnamoorthy, "Presage: Protecting structured address generation against soft errors," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, Dec. 2016, pp. 252–261.

[49] *LLVM*, https://llvm.org, 2019.

[50] *Clang*, https://clang.llvm.org, 2019.

[51] *Flang*, https://github.com/flang-compiler/flang, 2019.

[52] *Dragonegg*, https://dragonegg.llvm.org, 2019.

[53] *Libdwarf*, https://www.prevanders.net/dwarf.html, 2019.

[54] *Libffi*, https://sourceware.org/libffi/, 2019.

[55] *Google protobuf*, https://developers.google.com/protocol-buffers/, 2019.

[56] *Mhash*, http://mhash.sourceforge.net/, 2019.

[57] F. Long, S. Sidiroglou-Douskos, and M. Rinard, "Automatic runtime error repair and containment via recovery shepherding," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, Edinburgh, United Kingdom: ACM, 2014, pp. 227–238, ISBN: 978-1-4503-2784-8.

[58] C. Chen, G. Eisenhauer, M. Wolf, and S. Pande, "Ladr: Low-cost application-level detector for reducing silent output corruptions," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18, Tempe, Arizona: ACM, 2018, pp. 156–167, ISBN: 978-1-4503-5785-2.

[59] G. Georgakoudis, I. Laguna, D. S. Nikolopoulos, and M. Schulz, "Refine: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17, Denver, Colorado: ACM, 2017, 29:1–29:14, ISBN: 978-1-4503-5114-0.

[60] C.-K. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez, "Evaluating and accelerating high-fidelity error injection for hpc," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18, Dallas, Texas: IEEE Press, 2018, 45:1–45:13.

[61] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating bugs as allergies—a safe method to survive software failures," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05, Brighton, United Kingdom: ACM, 2005, pp. 235–248, ISBN: 1-59593-079-5.

[62] L. Guo and D. Li, "Moard: Modeling application resilience to transient faults on data objects," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 878–889.

[63] G. Upasani, X. Vera, and A. González, "Setting an error detection infrastructure with low cost acoustic wave detectors," *SIGARCH Comput. Archit. News*, vol. 40, no. 3, pp. 333–343, Jun. 2012.

[64] G. Upasani, X. Vera, and A. Gonzalez, "Avoiding core's due & sdc via acoustic wave detectors and tailored error containment and recovery," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 37–48, Jun. 2014.

[65] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed soft error detection and recovery to avoid due and sdc via tail-dmr," vol. 16, no. 2, Dec. 2016.

[66] M. A. de Kruijf, K. Sankaralingam, and S. Jha, "Static analysis and compiler design for idempotent processing," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, Beijing, China: Association for Computing Machinery, 2012, pp. 475–486, ISBN: 9781450312059.

[67] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler directed lightweight soft error resilience," *ACM Sigplan Notices*, vol. 50, no. 5, pp. 1–10, 2015.

[68] J. V. D. Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 17–32, ISBN: 978-1-931971-33-1.